# Synthesis and Verification Design Guide

XILINX®

"Xilinx" and the Xilinx logo shown above are registered trademarks of Xilinx, Inc. Any rights not expressly granted herein are reserved.

CoolRunner, RocketChips, Rocket IP, Spartan, StateBENCH, StateCAD, Virtex, XACT, XC2064, XC3090, XC4005, and XC5210 are registered trademarks of Xilinx, Inc.

The shadow X shown above is a trademark of Xilinx, Inc.

ACE Controller, ACE Flash, A.K.A. Speed, Alliance Series, AllianceCORE, Bencher, ChipScope, Configurable Logic Cell, CORE Generator, CoreLINX, Dual Block, EZTag, Fast CLK, Fast CONNECT, Fast FLASH, FastMap, Fast Zero Power, Foundation, Gigabit Speeds...and Beyond!, HardWire, HDL Bencher, IRL, J Drive, JBits, LCA, LogiBLOX, Logic Cell, LogiCORE, LogicProfessor, MicroBlaze, MicroVia, MultiLINX, NanoBlaze, PicoBlaze, PLUSASM, PowerGuide, PowerMaze, QPro, Real-PCI, RocketIO, SelectIO, SelectRAM, SelectRAM+, Silicon Xpresso, Smartguide, Smart-IP, SmartSearch, SMARTswitch, System ACE, Testbench In A Minute, TrueMap, UIM, VectorMaze, VersaBlock, VersaRing, Virtex-II Pro, Virtex-II EasyPath, Wave Table, WebFITTER, WebPACK, WebPOWERED, XABEL, XACT-Floorplanner, XACT-Performance, XACTstep Advanced, XACTstep Foundry, XAM, XAPP, X-BLOX +, XC designated products, XChecker, XDM, XEPLD, Xilinx Foundation Series, Xilinx XDTV, Xinfo, XSI, XtremeDSP and ZERO+ are trademarks of Xilinx, Inc.

The Programmable Logic Company is a service mark of Xilinx, Inc.

All other trademarks are the property of their respective owners.

Xilinx, Inc. does not assume any liability arising out of the application or use of any product described or shown herein; nor does it convey any license under its patents, copyrights, or maskwork rights or any rights of others. Xilinx, Inc. reserves the right to make changes, at any time, in order to improve reliability, function or design and to supply the best product possible. Xilinx, Inc. will not assume responsibility for the use of any circuitry described herein other than circuitry entirely embodied in its products. Xilinx provides any design, code, or information shown or described herein "as is." By providing the design, code, or information as one possible implementation of a feature, application, or standard, Xilinx makes no representation that such implementation is free from any claims of infringement. You are responsible for obtaining any rights you may require for your implementation. Xilinx expressly disclaims any warranty whatsoever with respect to the adequacy of any such implementation, including but not limited to any warranties or representations that the implementation is free from claims of infringement, as well as any implied warranties of merchantability or fitness for a particular purpose. Xilinx, Inc. devices and products are protected under U.S. Patents. Other U.S. and foreign patents pending. Xilinx, Inc. does not represent that devices shown or products described herein are free from patent infringement or from any other third party right. Xilinx, Inc. assumes no obligation to correct any errors contained herein or to advise any user of this text of any correction if such be made. Xilinx, Inc. will not assume any liability for the accuracy or correctness of any engineering or software support or assistance provided to a user.

Xilinx products are not intended for use in life support appliances, devices, or systems. Use of a Xilinx product in such applications without the written consent of the appropriate Xilinx officer is prohibited.

The following table shows the revision history for this document.

| | Version | Revision |
|---|---|---|
| 06/01/00 | 1.0 | Initial Xilinx® release. |
| 06/15/00 | 1.1 | Accumulated miscellaneous updates and bug fixes. |
| 07/26/00 | 1.2 | Accumulated miscellaneous updates and bug fixes. |
| 08/28/00 | 1.3 | Fine tuning of text frame and paragraph format spacings. |
| 04/11/01 | 2.0 | Revised formats to take advantage of FrameMaker 6.0 book features. |
| 05/02/01 | 2.1 | Master page changes. |
| 07/11/01 | 2.2 | Accumulated miscellaneous updates and bug fixes. |
| 04/04/02 | 2.21 | Updated trademarks page in **ug000_title.fm**. |
| 06/24/02 | 3.0 | Initial Xilinx® release of corporate-wide common template set, used for User Guides, Tutorials, Release Notes, Manuals, and other lengthy, multiple-chapter documents created by both CMP and ITP. See related documents for further information.<br>*Descriptions for revisions prior to v3.0 have been abbreviated.* For a full summary of revision changes prior to v3.0, refer to v2.21 template set. |
| 06/06/03 | 4.0 | Name changed from *Synthesis and Simulation Design Guide* to *Synthesis and Verification Design* guide.<br>Added new chapter for Equivalency Checking.<br>Accumulated miscellaneous updates and bug fixes. |

# *About This Guide*

This manual provides a general overview of designing Field Programmable Gate Arrays (FPGAs) with Hardware Description Languages (HDLs). It includes design hints for the novice HDL user, as well as for the experienced user who is designing FPGAs for the first time.

The design examples in this manual were created with Verilog and VHSIC Hardware Description Language (VHDL); compiled with various synthesis tools; and targeted for Spartan-II™, Spartan-IIE™, Spartan-3™, Virtex™, Virtex-E™, Virtex-II™, Virtex-II Pro™ and Virtex-II Pro X™ devices. Xilinx® equally endorses both Verilog and VHDL. VHDL may be more difficult to learn than Verilog and usually requires more explanation.

This manual does not address certain topics that are important when creating HDL designs, such as the design environment; verification techniques; constraining in the synthesis tool; test considerations; and system verification. Refer to your synthesis tool's reference manuals and design methodology notes for additional information.

Before using this manual, you should be familiar with the operations that are common to all Xilinx® software tools.

## Guide Contents

This book contains the following chapters.

- Chapter 1, "Introduction," provides a general overview of designing Field Programmable Gate Arrays (FPGAs) with HDLs. This chapter also includes installation requirements and instructions.
- Chapter 2, "Understanding High-Density Design Flow," provides synthesis and Xilinx® implementation techniques to increase design performance and utilization.
- Chapter 3, "General HDL Coding Styles," includes HDL coding hints and design examples to help you develop an efficient coding style.
- Chapter 4, "Architecture Specific Coding Styles for Spartan™-II / -3,Virtex™ / -E / -II / -II Pro / -II Pro X," includes coding techniques to help you use the latest Xilinx® devices.
- Chapter 5 "Virtex-II Pro™ Considerations," describes the special considerations encountered when simulating designs for Virtex-II Pro™ and Virtex-II Pro X™ FPGAs.
- Chapter 6, "Verifying Your Design," describes simulation methods for verifying the function and timing of your designs.
- Chapter 7, "Equivalency Checking," describes how to use third party tools to perform formal verification on your designs.

# Additional Resources

For additional information, go to http://support.xilinx.com. The following table lists some of the resources you can access from this website. You can also directly access these resources using the provided URLs.

| Resource | Description/URL |
|---|---|
| Tutorials | Tutorials covering Xilinx® design flows, from design entry to verification and debugging<br>http://support.xilinx.com/support/techsup/tutorials/index.htm |
| Answer Browser | Database of Xilinx® solution records<br>http://support.xilinx.com/xlnx/xil_ans_browser.jsp |
| Application Notes | Descriptions of device-specific design techniques and approaches<br>http://support.xilinx.com/apps/appsweb.htm |
| Data Sheets | Pages from The Programmable Logic Data Book, which contains device-specific information on Xilinx device characteristics, including readback, boundary scan, configuration, length count, and debugging<br>http://www.support.xilinx.com/xlnx/xweb/xil_publications_index.jsp |
| Problem Solvers | Interactive tools that allow you to troubleshoot your design issues<br>http://support.xilinx.com/support/troubleshoot/psolvers.htm |
| Tech Tips | Latest news, design tips, and patch information for the Xilinx® design environment<br>http://www.support.xilinx.com/xlnx/xil_tt_home.jsp |

# Conventions

This document uses the following conventions. An example illustrates each convention.

## Typographical

The following typographical conventions are used in this document:

| Convention | Meaning or Use | Example |
|---|---|---|
| `Courier font` | Messages, prompts, and program files that the system displays | `speed grade: - 100` |
| **`Courier bold`** | Literal commands that you enter in a syntactical statement | **`ngdbuild`** *`design_name`* |
| **Helvetica bold** | Commands that you select from a menu | **File → Open** |
|  | Keyboard shortcuts | **Ctrl+C** |

| Convention | Meaning or Use | Example |
|---|---|---|
| *Italic font* | Variables in a syntax statement for which you must supply values | **ngdbuild** *design_name* |
| | References to other manuals | See the *Development System Reference Guide* for more information. |
| | Emphasis in text | If a wire is drawn so that it overlaps the pin of a symbol, the two nets are *not* connected. |
| Square brackets [ ] | An optional entry or parameter. However, in bus specifications, such as **bus[7:0]**, they are required. | **ngdbuild** [*option_name*] *design_name* |
| Braces { } | A list of items from which you must choose one or more | **lowpwr ={on\|off}** |
| Vertical bar \| | Separates items in a list of choices | **lowpwr ={on\|off}** |
| Vertical ellipsis<br>.<br>.<br>. | Repetitive material that has been omitted | `IOB #1: Name = QOUT'`<br>`IOB #2: Name = CLKIN'`<br>.<br>.<br>. |
| Horizontal ellipsis . . . | Repetitive material that has been omitted | **allow block** *block_name loc1 loc2 ... locn;* |

## Online Document

The following conventions are used in this document:

| Convention | Meaning or Use | Example |
|---|---|---|
| Blue text | Cross-reference link to a location in the current file or in another file in the current document | See the section "Additional Resources"for details.<br>Refer to "Using Synthesis Tools" for details. |
| Red text | Cross-reference link to a location in another document | See Figure 2-5 in the *Virtex-II Platform FPGA User Guide.* |
| Blue, underlined text | Hyperlink to a website (URL) | Go to http://www.xilinx.com for the latest speed files. |

# Table of Contents

## Preface: About This Guide

## Chapter 1: Introduction

## Chapter 2: Understanding High-Density Design Flow

## Chapter 3: General HDL Coding Styles

## Chapter 4: Architecture Specific Coding Styles for Spartan™-II/ -3,Virtex™/-E/-II/-II Pro/-II Pro X

## Chapter 5: Virtex-II Pro™ Considerations

## Chapter 6: Verifying Your Design

## Chapter 7: Equivalency Checking

Chapter 1

# Introduction

This chapter provides a general overview of designing Field Programmable Gate Arrays (FPGAs) with HDLs, and also includes installation requirements and instructions. It includes the following sections.

- "Architecture Support"
- "Overview of Hardware Description Languages"
- "Advantages of Using HDLs to Design FPGAs"
- "Designing FPGAs with HDLs"
- "Xilinx® Internet Websites"
- "Vendor Support Sites"

## Architecture Support

The software supports the following architecture families in this release.

- Virtex™-II / -E
- Virtex™-II PRO and Virtex™-II PRO X
- Spartan™-II / -IIE / -3
- CoolRunner™ XPLA3 / -II / -IIS
- XC9500™ / XL / XV

## Overview of Hardware Description Languages

Hardware Description Languages (HDLs) are used to describe the behavior and structure of system and circuit designs. This chapter includes a general overview of designing FPGAs with HDLs. HDL design examples are provided in subsequent chapters of this book, and design examples can be downloaded from the Xilinx® website. System requirements and installation instructions for designs available from the web are also provided in this chapter.

This chapter also includes a brief description of why using FPGAs is more advantageous than using ASICs for your design needs.

To learn more about designing FPGAs with HDLs, Xilinx® recommends that you enroll in the appropriate training classes offered by Xilinx® and by the vendors of synthesis software. An understanding of FPGA architecture allows you to create HDL code that effectively uses FPGA system features.

For the latest information on Xilinx® parts and software, visit the Xilinx® website at http://www.xilinx.com. On the Xilinx® home page, click on Products. You can get answers to your technical questions from the Xilinx® support website at http://www.support.xilinx.com. On the support home page, click on Advanced Search to set up search criteria that match your technical questions. You can also download software service packs from from this website. On the support home page, click on Software, and then Service Packs. Software documentation, tutorials, and design files are also available from this website.

# Advantages of Using HDLs to Design FPGAs

Using HDLs to design high-density FPGAs is advantageous for the following reasons.

- *Top-Down Approach for Large Projects*—HDLs are used to create complex designs. The top-down approach to system design supported by HDLs is advantageous for large projects that require many designers working together. After the overall design plan is determined, designers can work independently on separate sections of the code.
- *Functional Simulation Early in the Design Flow*—You can verify the functionality of your design early in the design flow by simulating the HDL description. Testing your design decisions before the design is implemented at the RTL or gate level allows you to make any necessary changes early in the design process.
- *Synthesis of HDL Code to Gates*—You can synthesize your hardware description to a design implemented with gates. This step decreases design time by eliminating the need to define every gate. Synthesis to gates also reduces the number of errors that can occur during a manual translation of a hardware description to a schematic design. Additionally, you can apply the automation techniques used by the synthesis tool (such as machine encoding styles or automatic I/O insertion) during the optimization of your design to the original HDL code, resulting in greater efficiency.
- *Early Testing of Various Design Implementations*—HDLs allow you to test different implementations of your design early in the design flow. You can then use the synthesis tool to perform the logic synthesis and optimization into gates. Additionally, Xilinx® FPGAs allow you to implement your design at your computer. Since the synthesis time is short, you have more time to explore different architectural possibilities at the Register Transfer Level (RTL). You can reprogram Xilinx® FPGAs to test several implementations of your design.
- *Reuse of RTL Code* —You can retarget RTL code to new FPGA architectures with a minimum of recoding.

# Designing FPGAs with HDLs

If you are more familiar with schematic design entry, you may find it difficult at first to create HDL designs. You must make the transition from graphical concepts, such as block diagrams, state machines, flow diagrams, and truth tables, to abstract representations of design components. You can ease this transition by not losing sight of your overall design plan as you code in HDL. To effectively use an HDL, you must understand the syntax of the language; the synthesis and simulator software; the architecture of your target device; and the implementation tools. This section gives you some design hints to help you create FPGAs with HDLs.

## Using Verilog

Verilog® is popular for synthesis designs because it is less verbose than traditional VHDL, and it is standardized as IEEE-STD-1364-95. It was not originally intended as an input to synthesis, and many Verilog constructs are not supported by synthesis software. The

Verilog examples in this manual were tested and synthesized with current, commonly-used FPGA synthesis software. The coding strategies presented in the remaining chapters of this manual can help you create HDL descriptions that can be synthesized.

## Using VHDL

VHSIC Hardware Description Language (VHDL) is a hardware description language for designing Integrated Circuits (ICs). It was not originally intended as an input to synthesis, and many VHDL constructs are not supported by synthesis software. However, the high level of abstraction of VHDL makes it easy to describe the system-level components and testbenches that are not synthesized. In addition, the various synthesis tools use different subsets of the VHDL language. The examples in this manual will work with most commonly used FPGA synthesis software. The coding strategies presented in the remaining chapters of this manual can help you create HDL descriptions that can be synthesized.

## Comparing ASICs and FPGAs

Xilinx® FPGAs are reprogrammable and when combined with an HDL design flow can greatly reduce the design and verification cycle seen with traditional ASICs.

## Using Synthesis Tools

Most of the commonly-used FPGA synthesis tools have special optimization algorithms for Xilinx® FPGAs. Constraints and compiling options perform differently depending on the target device. There are some commands and constraints in ASIC synthesis tools that do not apply to FPGAs and, if used, may adversely impact your results. You should understand how your synthesis tool processes designs before creating FPGA designs. Most FPGA synthesis vendors include information in their manuals specifically for Xilinx® FPGAs.

## Using FPGA System Features

You can improve device performance and area utilization by creating HDL code that uses FPGA system features, such as global reset, wide I/O decoders, and memory. FPGA system features are described in this manual.

## Designing Hierarchy

Current HDL design methods are specifically written for ASIC designs. You can use some of these ASIC design methods when designing FPGAs; however, certain techniques may unnecessarily increase the number of gates or CLB levels. This design guide will train you in techniques for optional FPGA design methodologies.

Design hierarchy is important in the implementation of an FPGA and also during incremental or interactive changes. Some synthesizers maintain the hierarchical boundaries unless you group modules together. Modules should have registered outputs so their boundaries are not an impediment to optimization. Otherwise, modules should be as large as possible within the limitations of your synthesis tool. The “5,000 gates per module” rule is no longer valid, and can interfere with optimization. Check with your synthesis vendor for the current recommendations for preferred module size. As a last resort, use the grouping commands of your synthesizer, if available. The size and content of the modules influence synthesis results and design implementation. This manual describes how to create effective design hierarchy.

### Specifying Speed Requirements

To meet timing requirements, you should understand how to set timing constraints in both the synthesis and placement/routing tools. For more information, see "Setting Constraints" in Chapter 2.

## Xilinx® Internet Websites

You can get product information and product support from the Xilinx® internet websites. Both sites are described in the following sections.

### Xilinx® World Wide Web Site

You can reach the Xilinx® website at http://www.xilinx.com The following features can be accessed from the Xilinx® website.

- *Products* — You can find information about new Xilinx® products that are being offered, as well as previously announced Xilinx® products.
- *Service and Support* — You can jump to the Xilinx® technical support site by choosing Service and Support.
- *Xpresso Cafe* —You can purchase Xilinx® software, hardware and software tool education classes through Xilinx® and Xilinx distributors.

### Technical Support Website

Answers to questions, tutorials, Application notes, software manuals and information on using Xilinx® products can be found on the technical support website. You can reach the support website at http://www.support.xilinx.com. The following features can be accessed from the Xilinx® support website.

- *Troubleshoot* — You can do an advanced search on the Answers Database to troubleshoot questions or issues you have with your design.
- *Software* — You can download the latest software service packs, IP updates, and product information from the Xilinx Support Website.
- *Library* — You can view the Software manuals from this website. The manuals are provided in both HTML, viewable through most HTML browsers, and PDF. The Databook, CORE Generator™ documentation and data sheets are also available.
- *Design* — You can find helpful application notes that illustrate specific design solutions and methodologies.
- *Services* — You can open a support case when you need to have information from a Xilinx® technical support person. You can also find information about your hardware or software order.
- *Feedback* —We are always interested in how well we're serving our customers. You can let us know by filling out our customer service survey questionnaire.

You can contact Xilinx® technical support and application support for additional information and assistance in the following ways.

## Technical and Applications Support Hotlines

The telephone hotlines give you direct access to Xilinx® Application Engineers worldwide. You can also e-mail or fax your technical questions to the same locations.

*Table 1-1:* **Technical Support**

| Location | Telephone | Electronic Mail | Support Hours |
|---|---|---|---|
| North America | +1 800 255 7778 or<br>+1 408 879 5199 | link to http://support.xilinx.com/support/clearexpress/websupport.htm | M,T,W, F 7:00-5:00 PST<br>Th 7:00-4:00 PST |
| United Kingdom | +44-870-7350-610 | link to http://support.xilinx.com/support/clearexpress/websupport.htm | M-F 8:00 -5:30 GMT |
| France | +33-1-3463-0100 | link to http://support.xilinx.com/support/clearexpress/websupport.htm | M-F 8:00 -5:30 GMT |
| Germany | +49- 180-3-60-60-60 | link to http://support.xilinx.com/support/clearexpress/websupport.htm | M-F 8:00 -5:30 GMT |
| [a]Sweden | +46- 8-33-14-00 | link to http://support.xilinx.com/support/clearexpress/websupport.htm | M-F 8:00 -5:30 GMT |
| [a]Netherlands | +31 (0)800 0233868 | link to http://support.xilinx.com/support/clearexpress/websupport.htm | M-F 8:00 -5:30 GMT |
| [a]Belgium | +32 (0)800 90913 | link to http://support.xilinx.com/support/clearexpress/websupport.htm | M-F 8:00 -5:30 GMT |
| Rest of Europe | +44 870 7350-610 | link to http://support.xilinx.com/support/clearexpress/websupport.htm | M-F 8:00 -5:30 GMT |
| Japan | +81 3 5321 7750 | link to http://support.xilinx.com/support/clearexpress/websupport.htm | M-F 9:00 -12:00,<br>1:00-5:00 JST |
| Hong Kong | +852 2 424 5200 | china_support@xilinx.com | |
| Taiwan | +886 2 2174 1388 | taiwan_support@xilinx.com | |
| China | +852 2 424 5200 | china_support@xilinx.com | |

*Table 1-1:* **Technical Support**

| Location | Telephone | Electronic Mail | Support Hours |
|---|---|---|---|
| [a]Rest of Asia | +1 408 879 5199 | link to http://support.xilinx.com/support/clearexpress/websupport.htm | |
| Corporate Switchboard | 1-408-559-7778 | link to http://support.xilinx.com/support/clearexpress/websupport.htm | |

a. English language support

***Note:*** When e-mailing or faxing inquiries, provide your complete name, company name, and phone number. Also, provide a complete problem description including your design entry software and design stage.

### Xilinx® FTP Site

ftp://ftp.xilinx.com

The FTP site provides online access to automated tutorials, design examples, online documents, utilities, and published patches.

## Vendor Support Sites

Vendor support for synthesis and verification products can be obtained at the following locations.

*Table 1-2:* **Vendor Support Sites**

| Vendor Name | Telephone | Electronic Mail | Website |
|---|---|---|---|
| Synopsys® | 1-800-245-8005 | support_center@synopsys.com | www.synopsys.com |
| Cadence®-Concept-HDL | 1-877-237-4911 | support@cadence.com | sourcelink.cadence.com |
| Mentor Graphics® | 1-800-547-4303 | support_net@mentor.com | www.mentor.com |
| Synplicity® | 1-408-548-6000 | support@synplicity.com | www.synplicity.com |

XILINX®

*Chapter 2*

# *Understanding High-Density Design Flow*

This chapter describes the steps in a typical HDL design flow. Although these steps may vary with each design, the information in this chapter is a good starting point for any design. This chapter includes the following sections.

- "Design Flow"
- "Entering your Design and Selecting Hierarchy"
- "Functional Simulation of your Design"
- "Synthesizing and Optimizing your Design"
- "Setting Constraints"
- "Evaluating Design Size and Performance"
- "Evaluating your Design for Coding Style and System Features"
- "Modular Design and Incremental Design (ECO)"
- "Placing and Routing Your Design"
- "Timing Simulation of Your Design"

# Design Flow

An overview of the design flow steps is shown in the following figure

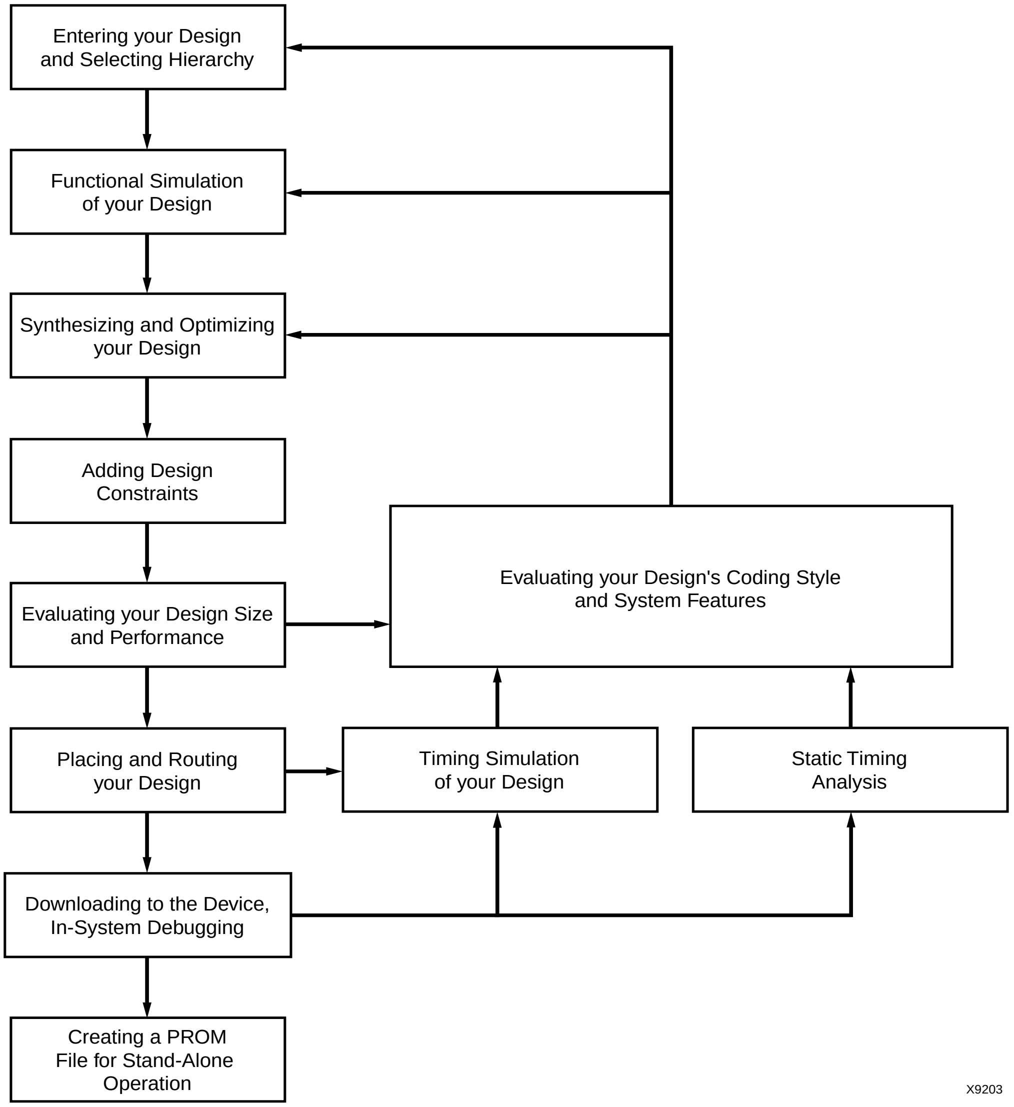

Entering your Design and Selecting Hierarchy
Functional Simulation of your Design
Synthesizing and Optimizing your Design
Adding Design Constraints
Evaluating your Design Size and Performance
Evaluating your Design's Coding Style and System Features
Placing and Routing your Design
Timing Simulation of your Design
Static Timing Analysis
Downloading to the Device, In-System Debugging
Creating a PROM File for Stand-Alone Operation
X9203

*Figure 2-1:* **Design Flow Overview**

# Entering your Design and Selecting Hierarchy

The first step in implementing your design is creating the HDL code based on your design criteria.

## Design Entry Recommendations

The following recommendations can help you create effective designs.

### Using RTL Code

By using register transfer level (RTL) code and avoiding (when possible) instantiating specific components, you can create designs with the following characteristics.

***Note:*** In some cases instantiating optimized CORE Generator™ or LogiCORE™ modules is beneficial with RTL.

- Readable code
- Faster and simpler simulation
- Portable code for migration to different device families
- Reusable code for future designs

### Carefully Select Design Hierarchy

Selecting the correct design hierarchy is advantageous for the following reasons.

- Improves simulation and synthesis results
- Improves debugging and modifying modular designs
- Allows parallel engineering (a team of engineers can work on different parts of the design at the same time)
- Improves the placement and routing of your design by reducing routing congestion and improving timing
- Allows for easier code reuse in the current design, as well as in future designs

## Architecture Wizard

The Architecture Wizard is a graphical application provided in Project Navigator that lets you configure complicated aspects of some Xilinx® devices. The Architecture Wizard consists of several components that you can use to configure specific device features. Each component is presented as an independent wizard. The following is a list of the wizards that make up the Architecture Wizard:

- Clocking Wizard
- RocketIO™ Wizard

The Architecture Wizard produces an XAW file, which is an XDM file with .xaw file extension. The Architecture Wizard can create a new XAW file, or it can read in an existing XAW file. When it reads in an existing XAW file, it allows you to modify the settings. When you finish with the wizard, the new data is saved to the same XAW file that was read in.

The Architecture Wizard can also produce a VHDL, Verilog, or EDIF file, depending on the flow type that is passed to it. The generated HDL output is a module (consisting of one or more primitives and the corresponding properties) and not just a code snippet. This allows the output file to be referenced from the HDL Editor. There is no UCF output file, since the necessary attributes are embedded inside the HDL file.

Launch the Architecture Wizard from Project Navigator. From the File dropdown menu, select **Project** → **New Source** → **Architecture Wizard...** menu item.

### Clocking Wizard

The Clocking Wizard component of the Architecture Wizard provides the following functions.

- Specifies Setup information.
- Lets you view the DCM component, specify attributes, generate corresponding components and signals, and execute DRC checks.
- Displays up to eight clock buffers.
- Sets up the Feedback Path information.
- Sets up the Clock Frequency Generator information and execute DRC checks.
- Lets you view and edit component attributes.
- Lets you view and edit component constraints.
- Automatically places one component in the XAW file.
- Saves component settings in a VHDL file.
- Saves component settings in a Verilog file.

### RocketIO™ Wizard

The RocketIO™ Wizard component of the Architecture Wizard provides the following functions.

- Provides the ability to specify RocketIO™ type.
- Provides the ability to define Channel Bonding options.
- Provides the ability to specify General Transmitter Settings including encoding, CRC and clock.
- Provides the ability to specify General Receptor Settings including encoding, CRC and clock.
- Provides the ability to specify Synchronization.
- Provides the ability to specify Equalization, Signal integrity tip (resister, termination mode...).
- Provides the ability to view and edit component attributes.
- Provides the ability to view and edit component constraints.
- Provides the ability to automatically place one component in the XAW file.
- Provides the ability to save component settings to a VHDL file.
- Provides the ability to save component settings to a Verilog file.

## CORE Generator™

The CORE Generator™ System is a design tool that delivers parameterized cores optimized for Xilinx® FPGAs. It provides you with a catalog of ready-made functions ranging in complexity from simple arithmetic operators such as adders, accumulators, and multipliers, to system-level building blocks such as filters, transforms, FIFOs, and memories.

For each core it generates, the CORE Generator™ System produces an Electronic Data Interchange Format (EDIF) netlist (EDN file), a Verilog template (VEO) file with a Verilog (V) wrapper file, and / or a VHDL template (VHO) file with a VHDL (VHD) wrapper file. It may also create one or more NGC and NDF files. NGC files are produced for certain cores only.

The Electronic Data Netlist (EDN) and NGC files contain the information required to implement the module in a Xilinx® FPGA. Since NGC files are in binary format, ASCII NDF files may also be produced to communicate resource and timing information for NGC files to third party synthesis tools. The ASY and XSF symbol information files allow you to integrate the CORE Generator™ module into a schematic design for Mentor or ISE tools. VEO and VHO template files contain code that can be used as a model for instantiating a CORE Generator™ module in a Verilog or VHDL design. Finally, V and VHD wrapper files are provided to support functional simulation. These files contain simulation model customization data that is passed to a parameterized simulation model for the core. In the case of Verilog designs, the V wrapper file also provides the port information required to integrate the core into a Verilog design for synthesis.

For details about using CORE Generator™, see the *CORE Generator Guide*.

***Note:*** The V and VHD wrapper files generated by CORE Generator™ cores are provided mainly to support simulation and are not synthesizable.

# Functional Simulation of your Design

Use functional or RTL simulation to verify the syntax and functionality of your design. Use the following recommendations when simulating your design.

- Typically with larger hierarchical HDL designs, you should perform separate simulations on each module before testing your entire design. This makes it easier to debug your code.
- Once each module functions as expected, create a test bench to verify that your entire design functions as planned. You can use the test bench again for the final timing simulation to confirm that your design functions as expected under worst-case delay conditions.

# Synthesizing and Optimizing your Design

This section includes recommendations for compiling your designs to improve your results and decrease the run time.

***Note:*** Refer to your synthesis tool documentation for more information on compilation options and suggestions.

## Creating an Initialization File

Most synthesis tools provide a default initialization with default options. You may modify the initialization file or use the GUI to change compiler defaults, and to point to the applicable implementation libraries. Refer to your synthesis tool documentation for more information.

## Creating a Compile Run Script

FPGA Compiler II™, LeonardoSpectrum™, and Synplify™ all support TCL scripting. Using TCL scripting can make compiling your design easier and faster while achieving shorter compile times. With more advanced scripting you can run a compile multiple times using different options and write to different directories. You can also invoke and run other command line tools. The following are some sample scripts that can be run from the command line or from the GUI.

## FPGA Compiler II™

FPGA Scripting Tool (FST) implements a TCL-based command line interface for FPGA Compiler II™. FST can be accessed from a command line by typing the following.

- For FPGA Compiler II™

  **`fc2_shell`** -*f synth_file.tcl*

The script executes and puts you back at the UNIX or DOS prompt.

### FPGA Compiler II™ FST Example

The following FST commands can be run in FPGA Compiler II™.

- To create the project, enter the following.

  **`create_project`** -*dir . d_register*
- To open the project, enter the following.

  **`open_project`** *d_register*
- To add the files to the project, enter the following.

  **`add_file`** -*format VHDL ../src/d_register.vhd*
- To analyze the design, files enter the following.

  **`analyze_file`** -*progress*
- To create a chip for a device, enter the following.

  **`create_chip`** -*progress* -*target Virtex* -*device v50PQ240* -*speed* -*5* -*name d_register d_register*
- To set the top level as the current design, enter the following.

  **`current_chip`** *d_register*
- To optimize the design, enter the following.

  **`set opt_chip [format "%s-Optimized" d_register]`**

  **`optimize_chip`** -*progress* -*name $opt_chip*
- To write out the messages, enter the following.

  **`list_message`**
- To write out the netlist, enter the following.

  **`export_chip -`***progress* -*dir .*
- **`close_project`**
- **`quit`**

## LeonardoSpectrum™

You can run the following TCL script from LeonardoSpectrum™ by doing one of the following.

1. Select the **File** → **Run Script** menu item from the LeonardoSpectrum™ graphical user interface.
2. Type in the Level 3 GUI command line, **`source`** *script_file.tcl*

3. Type in the UNIX/DOS prompt with the EXEMPLAR environment path set up, **spectrum** *–file script_file.tcl*
4. Type **spectrum** at the UNIX/DOS prompt. This puts you in a TCL prompt. Then at the TCL prompt type **source** *script_file.tcl*

### LeonardoSpectrum™ TCL Examples

You can enter the following TCL commands in LeonardoSpectrum™.

- To set the part type, enter the following.

  **set part** *v50ecs144*

- To read the HDL files, enter the following.

  **read** *macro1.vhd macro2.vhd top_level.vhd*

- To set assign buffers, enter the following.

  **PAD** *IBUF_LVDS data(7:0)*

- To optimize while preserving hierarchy, enter the following.

  **optimize** -*ta xcve* -*hier preserve*

- To write out the EDIF file, enter the following.

  **auto_write** *./M1/ff_example.edf*

## Synplify™

You can run the following TCL script from Synplify™ by doing one of the following:

1. Use the **File** → **Run TCL Script** menu item from the GUI.
2. Type **synplify** -*batch script_file.tcl* at a UNIX/DOS command prompt.

### Synplify™ TCL Example

You can enter the following TCL commands in Synplify™.

- To start a new project, enter the following.

  **project** -*new*

- To set device options, enter the following.

  **set_option** -*technology Virtex-E*

  **set_option** -*part XCV50E*

  **set_option** -*package CS144*

  **set_option** -*speed_grade* -*8*

- To add file options, enter the following.

  **add_file** -*constraint “watch.sdc”*

  **add_file** -*vhdl* -*lib work “macro1.vhd”*

  **add_file** -*vhdl* -*lib work “macro2.vhd”*

  **add_file** -*vhdl* -*lib work “top_levle.vhd”*

- To set compilation / mapping options, enter the following.

  ```
  set_option -default_enum_encoding onehot
  set_option -symbolic_fsm_compiler true
  set_option -resource_sharing true
  ```

- To set simulation options, enter the following.

  ```
  set_option -write_verilog false
  set_option -write_vhdl false
  ```

- To set automatic place and route (vendor) options, enter the following.

  ```
  set_option -write_apr_constraint true
  set_option -part XCV50E
  set_option -package CS144
  set_option -speed_grade -8
  ```

- To set result format / file options, enter the following.

  ```
  project -result_format “edif”
  project -result_file “top_level.edf”
  project -run
  project -save “watch.prj”
  ```

- **`exit`**

## Compiling Your Design

Use the recommendations in this section to successfully compile your design.

### Modifying your Design

You may need to modify your code to successfully compile your design because certain design constructs that are effective for simulation may not be as effective for synthesis. The synthesis syntax and code set may differ slightly from the simulator syntax and code set.

### Compiling Large Designs

Older versions of synthesis tools required incremental design compilations to decrease run times. Some or all levels of hierarchy were compiled with separate compile commands and saved as output or database files. The output netlist or compiled database file for each module was read during synthesis of the top level code. This method is not necessary with new synthesis tools, which can handle large designs from the top down. The 5,000 gates per module rule of thumb no longer applies with the new synthesis tools. Refer to your synthesis tool documentation for details.

### Saving Compiled Design as EDIF

After your design is successfully compiled, save it as an EDIF file for input to the Xilinx® software.

## Reading COREs

Synplify Pro™, LeonardoSpectrum™ and XST support the reading in of CORE Generator™ EDIF files for timing and area analysis. When these tools read in the EDIF files better timing optimizations can be done since the delay through the logic elements inside of the CORE file will be known. The procedure for reading in COREs in these synthesis tools are as follows:

- XST

  Invoke XST using the read_cores switch. When the switch is set to on, the default, XST, reads in EDIF and NGC netlists. Please refer to the *XST User Guide* for more information.

- LeonardoSpectrum™

  Use the read_coregen TCL command line option. Please refer to Answer Record 13159 for more information.

- Synplify Pro™

  EDIF is treated as just another source format, but when reading in EDIF, you must specify the top level VHDL/Verilog in your project. Support for reading in EDIF is included in Symplify Pro version 7.3. Please see the Symplify documentation for more information.

# Setting Constraints

You can define timing specifications for your design in the User Constraints File (UCF). You can use the Xilinx® Constraints Editor which provides a graphical user interface allowing for easy constraints specification. You can also enter constraints directly into the UCF file. Both methods are described in the following sections. Most synthesis tools support an easy to use Constraints Editor interface for entering constraints in your design.

## Using the UCF File

The UCF gives you tight control of the overall specifications by giving you access to more types of constraints; the ability to define precise timing paths; and the ability to prioritize signal constraints. Furthermore, you can group signals together to simplify timing specifications. Some synthesis tools translate certain synthesis constraints to Xilinx® implementation constraints. The translated constraints are placed in the NCF/EDIF file (NGC file for XST). For more information on timing specifications in the UCF file, refer to the *Constraints Guide* and the Answers Database on the Xilinx® Support Website.

## Using the Xilinx® Constraints Editor

The Xilinx® Constraints Editor is a GUI based tool that can be accessed from the Processes for Current Source window of the Project Navigator GUI (**Design Entry Utilities →User Constraints →Constraints Editor**), or from the command line (**`constraints_editor`**). The Constraints Editor enables you to easily enter design constraints in a spreadsheet form and writes out the constraints in the UCF file. This eliminates the need to know the UCF file syntax. The other benefit is the Constraints Editor reads the design and lists all the nets and elements in the design. This is very helpful in the HDL flow when the synthesis tool creates the names.

Some constraints are not available through the Constraints Editor. The unavailable constraints must be entered directly in the UCF file using a text editor. The new UCF file needs to be re-run through the Translate step or NGDBuild using the command line method. For more information on using the Xilinx® Constraints Editor, please refer to the *Constraints Editor Guide*.

### Using Synthesis Tools' Constraints Editor

The FPGA Compiler II™, LeonardoSpectrum™ and Synplify™ synthesis tools all have constraint editors to apply constraints to your HDL design. Refer to your synthesis tool's documentation for information on how to use the constraints editor specific to your synthesis environment. You can add the following constraints:

- Clock frequency or cycle and offset
- Input and Output timing
- Signal Preservation
- Module constraints
- Buffering ports
- Path timing
- Global timing

Generally, the timing constraints are written to an NCF file, and all other constraints are written to the output EDIF file. In XST, all constraints are written to the NGC file. Please refer to the documentation for your synthesis tool to obtain more information on Constraint Editors.

## Evaluating Design Size and Performance

Your design should meet the following requirements.

- Design must function at the specified speed
- Design must fit in the targeted device

After your design is compiled, you can determine preliminary device utilization and performance with your synthesis tool's reporting options. After your design is mapped by the Xilinx® tools, you can determine the actual device utilization. At this point in the design flow, you should verify that your chosen device is large enough to incorporate any future changes or additions, and that your design will perform as specified.

### Using your Synthesis Tool to Estimate Device Utilization and Performance

Use your synthesis tool's area and timing reporting options to estimate device utilization and performance. After compiling, use the report area command to obtain a report of device resource utilization. Some synthesis tools provide area reports automatically. Refer to your synthesis tool documentation for correct command syntax.

The device utilization and performance report lists the compiled cells in your design, as well as information on how your design is mapped in the FPGA. These reports are generally accurate because the synthesis tool creates the logic from your code and maps your design into the FPGA. However, these reports are different for the various synthesis tools. Some reports specify the minimum number of CLBs required, while other reports specify the "unpacked" number of CLBs to make an allowance for routing. For an accurate comparison, compare reports from the Xilinx® place and route tool after implementation. Also, any instantiated components, such as CORE Generator™ modules, EDIF files, or other components that your synthesis tool does not recognize during compilation, are not included in the report file. If you include these components in your design, you must include the logic area used by these components when estimating design size. Also, sections of your design may get trimmed during the mapping process, and may result in a smaller design.

### Using the Timing Report Command

Use your synthesis tool's timing report command to obtain a report with estimated data path delays. Refer to your synthesis vendor's documentation for command syntax.

The timing report is based on the logic level delays from the cell libraries and estimated wire-load models for your design. This report is an estimate of how close you are to your timing goals; however, it is not the actual timing for your design. An accurate report of your design's timing is only available after your design is placed and routed. This timing report does not include information on any instantiated components, such as CORE Generator™ modules, EDIF files, or other components that are not recognized by your synthesis tool during compilation.

## Determining Actual Device Utilization and Pre-routed Performance

To determine if your design fits the specified device, you must map it with the Xilinx® Map program. The generated report file *design_name*.mrp contains the implemented device utilization information. The report file can be read by double-clicking on **Map Report** in the Project Navigator Process Window. You can run the Map program from Project Navigator or from the command line.

### Using Project Navigator to Map Your Design

Use the following steps to map your design using Project Navigator.

***Note:*** For more information on using the Project Navigator, see *Project Navigator Online Help.*

1. After opening Project Navigator and creating your project, go to the Process Window and click the "+" symbol in front of **Implement Design**.
2. To run the Xilinx® Map program, double-click **Map.**
3. To view the Map Report, double-click **Map Report** in the Process Window. If the report does not currently exist, it is generated at this time. If a green check mark is in front of the report name, the report is up-to-date and no processing is performed. If the desired report is not up-to-date, you can click the report name and then select **Process** → **ReRun** to update the report before you view it. The auto-make process automatically runs only the necessary processes to update the report before displaying it. Or, you can select **Process** → **Rerun All** to re-run all processes— even those processes that are currently up-to-date— from the top of the design to the stage where the report would be.
4. View the Logic Level Timing Report with the Report Browser. This report shows the performance of your design based on logic levels and best-case routing delays.
5. At this point, you may want to start the Timing Analyzer to create a more specific report of design paths.
6. Use the Logic Level Timing Report and any reports generated with the Timing Analyzer or the Map program to evaluate how close you are to your performance and utilization goals. Use these reports to decide whether to proceed to the place and route phase of implementation, or to go back and modify your design or implementation options to attain your performance goals. You should have some slack in routing delays to allow the place and route tools to successfully complete your design. Use the

verbose option in the Timing Analyzer to see block-by-block delay. The timing report of a mapped design (before place and route) shows block delays, as well as minimum routing delays.

***Note:*** A typical Virtex™ /E/II/II Pro/II Pro X design should allow 40% of the delay for logic, and 60% of the delay for routing. If most of your time is taken by logic, then most likely, the design will not meet timing after place and route.

## Using the Command Line to Map Your Design

1. Translate your design as follows.

   **ngdbuild –p** *target_device design_name*.**edf**

2. Map your design as follows.

   **map** *design_name*.**ngd**

3. Use a text editor to view the Device Summary section of the *design_name*.mrp Map Report. This section contains the device utilization information.
4. Run a timing analysis of the logic level delays from your mapped design as follows.

   **trce [options]** *design_name*.**ncd**

   ***Note:*** For available options, enter only the trce command at the command line without any arguments.

   Use the Trace reports to evaluate how close you are to your performance goals. Use the report to decide whether to proceed to the place and route phase of implementation, or to go back and modify your design or implementation options to attain your performance goals. You should have some slack in routing delays to allow the place and route tools to successfully complete your design.

   The following is the Design Summary section of a Map Report containing device information.

```
Release 6.1i Map G.14
Xilinx Mapping Report File for Design ’tri_state_test’

Design Information
------------------
Command Line   : /build/bcxfndry/G.14/rtf/bin/sol/map -quiet -p xc2v40-cs144-6 -cm area -pr
b -k 4 -c 100 -tx off -o tri_state_test_map.ncd tri_state_test.ngd tri_state_test.pcf
Target Device  : x2v40
Target Package : cs144
Target Speed   : -6
Mapper Version : virtex2 -- $Revision: 1.8 $
Mapped Date    : Fri Feb 28 12:48:37 2003

Design Summary
--------------
Number of errors:      0
Number of warnings:    0
Logic Utilization:
   Number of 4 input LUTs:                         1 out of     512    1%
Logic Distribution:
   Number of occupied Slices:                      1 out of     256    1%
   Number of Slices containing only related logic: 1 out of       1  100%
   Number of Slices containing unrelated logic:    0 out of       1    0%
  *See NOTES below for an explanation of the effects of unrelated logic
Total Number 4 input LUTs:              1 out of     512    1%

   Number of bonded IOBs:               4 out of      88    4%
      IOB Flip Flops:                                  2
   Number of GCLKs:                     1 out of      16    6%

Total equivalent gate count for design:  28
Additional JTAG gate count for IOBs:  192
Peak Memory Usage:  45 MB

NOTES:

  Related logic is defined as being logic that shares connectivity -
  e.g. two LUTs are "related" if they share common inputs.
  When assembling slices, Map gives priority to combine logic that
  is related. Doing so results in the best timing performance.

  Unrelated logic shares no connectivity. Map will only begin
  packing unrelated logic into a slice once 99% of the slices are
  occupied through related logic packing.

  Note that once logic distribution reaches the 99% level through
  related logic packing, this does not mean the device is completely
  utilized. Unrelated logic packing will then begin, continuing until
  all usable LUTs and FFs are occupied. Depending on your timing
  budget, increased levels of unrelated logic packing may adversely
  affect the overall timing performance of your design.
```

```
Table of Contents
-----------------
Section 1 - Errors
Section 2 - Warnings
Section 3 - Informational
Section 4 - Removed Logic Summary
Section 5 - Removed Logic
Section 6 - IOB Properties
Section 7 - RPMs
Section 8 - Guide Report
Section 9 - Area Group Summary
Section 10 - Modular Design Summary

Section 1 - Errors
------------------
Section 2 - Warnings
--------------------
Section 3 - Informational
-------------------------
INFO:LIT:95 - All of the external outputs in this design are using slew rate
  limited output drivers. The delay on speed critical outputs can be
  dramatically reduced by designating them as fast outputs in the schematic.
INFO:MapLib - No environment variables are currently set.
INFO:MapLib:535 - The following Virtex BUFG(s) is/are being retargetted to
  Virtex2 BUFGMUX(s) with input tied to I0 and Select pin tied to constant 0:
  BUFGP symbol "clk_bufgp" (output signal=clk_bufgp)
Section 4 - Removed Logic Summary
---------------------------------
Section 5 - Removed Logic
-------------------------
Section 6 - IOB Properties
--------------------------

+----------------------------------------------------------------------------------------+
| IOB Name   | Type   |Direction|IO Standard|Drive   | Slew | Reg (s) | Resistor | IOB   |
|            |        |         |           |Strength| Rate |         |          | Delay |
+----------------------------------------------------------------------------------------+
| bidir      | IOB    | BIDIR   | LVTTL     | 12     | SLOW | OFF1    |          |       |
|            |        |         |           |        |      | ENFF1   |          |       |
| clk        | IOB    | INPUT   | LVTTL     |        |      |         |                  |
| d1         | IOB    | INPUT   | LVTTL     |        |      |         |          |       |
| en         | IOB    | INPUT   | LVTTL     |        |      |         |          |       |
+----------------------------------------------------------------------------------------+
Section 7 - RPMs
----------------
Section 8 - Guide Report
------------------------
Guide not run on this design.
Section 9 - Area Group Summary
------------------------------
No area groups were found in this design.
Section 10 - Modular Design Summary
-----------------------------------
Modular Design not used for this design.
```

# Evaluating your Design for Coding Style and System Features

At this point, if you are not satisfied with your design performance, you can re-evaluate your code and make any necessary improvements. Modifying your code and selecting different compiler options can dramatically improve device utilization and speed.

## Tips for Improving Design Performance

This section includes ways of improving design performance by modifying your code and by incorporating FPGA system features. Most of these techniques are described in more detail in this manual.

### Modifying Your Code

You can improve design performance with the following design modifications.

- Reduce levels of logic to improve timing
- Redefine hierarchical boundaries to help the compiler optimize design logic
- Pipeline
- Logic replication
- Use of CORE Generator™ modules
- Resource sharing
- Restructure logic

### Using FPGA System Features

After correcting any coding style problems, use any of the following FPGA system features in your design to improve resource utilization and to enhance the speed of critical paths.

Each device family has a unique set of system features. Go to the Xilinx® support website, http://support.xilinx.com, and refrence the Data Sheet for the system features available for the device you are targeting.

- Use clock enables
- In Virtex™ family components, modify large multiplexers to use tristate buffers
- Use one-hot encoding for large or complex state machines
- Use I/O registers when applicable
- In Virtex™ families, use dedicated shift registers
- In Virtex-II™ families, use dedicated multipliers

### Using Xilinx®-specific Features of Your Synthesis Tool

Most synthesis tools have special options for the Xilinx®-specific features listed in the previous section. Refer to your synthesis tool white papers, application notes, documentation and online help for detailed information on using Xilinx®-specific features.

# Modular Design and Incremental Design (ECO)

For information on Incremental Design (ECO), please refer to the following Application Notes:

- XAPP165: *"Using Xilinx and Exemplar for Incremental Designing (ECO)"*, v1.0 (8/9/99).
- XAPP164: *"Using Xilinx and Synplify for Incremental Designing (ECO)"*, v1.0 (8/6/99).

Xilinx® Development Systems feature Modular Design to help you plan and manage large designs. Reference the following link for more information on the Xilinx® Modular Design feature.

# Placing and Routing Your Design

***Note:*** For more information on placing and routing your design, refer to the *Development System Reference Guide*.

The overall goal when placing and routing your design is fast implementation and high-quality results. However, depending on the situation and your design, you may not always accomplish this goal, as described in the following examples.

- Earlier in the design cycle, run time is generally more important than the quality of results, and later in the design cycle, the converse is usually true.
- During the day, you may want the tools to quickly process your design while you are waiting for the results. However, you may be less concerned with a quick run time, and more concerned about the quality of results when you run your designs for an extended period of time (during the night or weekend).
- If the targeted device is highly utilized, the routing may become congested, and your design may be difficult to route. In this case, the placer and router may take longer to meet your timing requirements.
- If design constraints are rigorous, it may take longer to correctly place and route your design, and meet the specified timing.

## Decreasing Implementation Time

The options you select for the placement and routing of your design directly influence the run time. Generally, these options decrease the run time at the expense of the best placement and routing for a given device. Select your options based on your required design performance.

***Note:*** If you are using the command line, the appropriate command line option is provided in the following procedure.

Use the following steps to decrease implementation time in the Project Navigator. For details on implementing your design in Project Navigator see *Project Navigator Online Help*.

1. In the Project Navigator Process Window, right click **Place & Route** and then select **Properties.**

   The Process Properties dialog box appears.

   Set options in this dialog box as follows.

   ♦ Place & Route Effort Level

     Generally, you can reduce placement times by selecting a less CPU-intensive algorithm for placement. You can set the placement level at one of five settings from Lowest (fastest run time) to Highest (best results) with the default equal to Low. Use the –l switch at the command line to perform the same function.

***Note:*** In some cases, poor placement with a lower placement level setting can result in longer route times.

- Router Options

  You can limit router iterations to reduce routing times by setting the Number of Routing Passes option. However, this may prevent your design from meeting timing requirements, or your design may not completely route. From the command line, you can control router passes with the –i switch.

- Use Timing Constraints During Place and Route

  You can improve run times by not specifying some or all timing constraints. This is useful at the beginning of the design cycle during the initial evaluation of the placed and routed circuit. To disable timing constraints in the Project Navigator, uncheck the Use Timing Constraints check box. To disable timing constraints at the command line, use the –x switch with PAR.

2. Click **OK** to exit the Process Properties dialog box.
3. Double click **Place & Route** in the Process Window of Project Navigator to begin placing and routing your design.

## Improving Implementation Results

You can select options that *increase* the run time, but produce a better design. These options generally produce a faster design at the cost of a longer run time. These options are useful when you run your designs for an extended period of time (overnight or over the weekend). You can use the following options to improve implementation results. Detailed information for these options can be found in the *Development System Reference Guide*.

### Map Timing Option

Use the Xilinx® Map program Timing option to improve timing during the mapping phase. This switch directs the mapper to give priority to timing critical paths during packing. To use this feature at the command line, use the –timing switch. See the *Development System Reference Guide* for more information.

### Extra Effort Mode in PAR

Use the Xilinx® PAR program Extra Effort mode to invoke advanced algorithmic techniques to provide higher quality results. To use this feature at the command line, use the *–xe level* switch. The level can be a value from 0 to 5; the default is 1. Using level 0 turns off all extra effort off, and can significantly increase runtime. See the *Development System Reference Guide* for more information.

### Multi-Pass Place and Route

Use this feature to place and route your design with several different cost tables (seeds) to find the best possible placement for your design. This optimal placement results in shorter routing delays and faster designs. This works well when the router passes are limited (with the –i option). After an optimal cost table is selected, use the reentrant routing feature to finish the routing of your design. To use this feature double-click on **Multi Pass Place & Route** in the Process Window of Project Navigator, or specify this option at the command line with the –n switch. See the *Development System Reference Guide* for a description of Multi-Pass Place and Route, and how to set the appropriate options.

## Turns Engine Option (UNIX only)

This option is a Unix-only feature that works with the Multi-Pass Place and Route option to allow parallel processing of placement and routing on several Unix machines. The only limitation to how many cost tables are concurrently tested is the number of workstations you have available. To use this option in Project Navigator, see the *Project Navigator Online Help* for a description of the options that can be set under Multi-Pass Place and Route. To use this feature at the command line, use the –m switch to specify a node list, and the –n switch to specify the number of place and route iterations.

***Note:*** For more information on the turns engine option, refer to the *Development System Reference Guide*.

## Reentrant Routing Option

Use the reentrant routing option to further route an already routed design. The router reroutes some connections to improve the timing or to finish routing unrouted nets. You must specify a placed and routed design (.ncd) file for the implementation tools. This option is best used when router iterations are initially limited, or when your design timing goals are close to being achieved.

### From the Project Navigator

To initiate a reentrant route from Project Navigator, follow these steps. See the *Project Navigator Online Help* for details on reentrant routing.

1. In the Project Navigator Process Window, right click **Place & Route** and then select **Properties.**

   The Process Properties dialog box appears. Set the Place and Route Mode option to **Reentrant Route.**

2. Click **OK** to exit the Process Properties dialog box.
3. Double click **Place & Route** in the Process Window of Project Navigator to begin placing and routing your design.

### Using PAR and Cost Tables

The PAR module places in two stages: a constructive placement and an optimizing placement. PAR writes the NCD file after constructive placement and modifies the NCD after optimizing placement.

During constructive placement, PAR places components into sites based on factors such as constraints specified in the input file (for example, certain components must be in certain locations), the length of connections, and the available routing resources. This placement also takes into account "cost tables," which assign weighted values to each of the relevant factors. There are 100 possible cost tables. Constructive placement continues until all components are placed. PAR writes the NCD file after constructive placement.

For more information on PAR and Cost Tables, refer to the *Development System Reference Guide.*

#### From the Command Line

To initiate a reentrant route from the command line, you can run PAR with the –k and –p options, as well as any other options you want to use for the routing process. You must either specify a unique name for the post reentrant routed design (.ncd) file or use the –w switch to overwrite the previous design file, as shown in the following examples.

```
par -k -p other_options design_name.ncd new_name.ncd
par -k -p -w other_options design_name.ncd design.ncd
```

### Guide Option

This option is generally not recommended for synthesis-based designs, except for modular design flows. Re-synthesizing modules can cause the signal and instance names in the resulting netlist to be significantly different from those in earlier synthesis runs. This can occur even if the source level code (Verilog or VHDL) contains only a small change. Because the guide process is dependent on the names of signals and comps, synthesis designs often result in a low match rate during the guiding process. Generally, this option does not improve implementation results.

For information on guide in modular design flows, refer to the Xilinx® Modular Design web page.

# Timing Simulation of Your Design

***Note:*** Refer to Chapter 6, "Verifying Your Design" for more information on design simulation.

Timing simulation is important in verifying the operation of your circuit after the worst-case placed and routed delays are calculated for your design. In many cases, you can use the same test bench that you used for functional simulation to perform a more accurate simulation with less effort. You can compare the results from the two simulations to verify that your design is performing as initially specified. The Xilinx® tools create a VHDL or Verilog simulation netlist of your placed and routed design, and provide libraries that work with many common HDL simulators.

## Timing Analysis Using TRACE

Timing-driven PAR is based upon the Xilinx® timing analysis software, an integrated static timing analysis tool (that is, it does not depend on input stimulus to the circuit). This means that placement and routing are executed according to timing constraints that you specify in the beginning of the design process. The timing analysis software interacts with PAR to ensure that the timing constraints you impose on the design are met.

If you have timing constraints, TRACE generates a report based on your constraints. If there are no constraints, the timing analysis tool has an option to write out a timing report containing the following.

- An analysis that enumerates all clocks and the required OFFSETs for each clock.
- An analysis of paths having only combinatorial logic, ordered by delay.

For more information on TRACE, refer to the *Development System Reference Guide*. For more information on Timing Analysis, refer to the *Timing Analyzer Online Help*.

Chapter 3

# General HDL Coding Styles

This chapter contains HDL coding styles and design examples to help you develop an efficient coding style. It includes the following sections.

- "Naming and Labeling Styles"
- "Specifying Constants"
- "Choosing Data Type (VHDL only)"
- "Coding for Synthesis"

HDLs contain many complex constructs that are difficult to understand at first. Also, the methods and examples included in HDL manuals do not always apply to the design of FPGAs. If you currently use HDLs to design ASICs, your established coding style may unnecessarily increase the number of gates or CLB levels in FPGA designs.

HDL synthesis tools implement logic based on the coding style of your design. To learn how to efficiently code with HDLs, you can attend training classes, read reference and methodology notes, and refer to synthesis guidelines and templates available from Xilinx® and the synthesis vendors. When coding your designs, remember that HDLs are mainly hardware description languages. You should try to find a balance between the quality of the end hardware results and the speed of simulation.

The coding hints and examples included in this chapter are not intended to teach you every aspect of VHDL or Verilog, but they should help you develop an efficient coding style.

## Naming and Labeling Styles

Because HDL designs are often created by design teams, Xilinx® recommends that you agree on a style for your code at the beginning of your project. An established coding style allows you to read and understand code written by your fellow team members. Also, inefficient coding styles can adversely impact synthesis and simulation, which can result in slow circuits. Additionally, because portions of existing HDL designs are often used in new designs, you should follow coding standards that are understood by the majority of HDL designers. This section of the manual provides a list of suggested coding styles that you should establish before you begin your designs.

### Using Xilinx® Naming Conventions

Use the Xilinx® naming conventions listed in this section for naming signals, variables, and instances that are translated into nets, buses, and symbols.

***Note:*** Most synthesis tools convert illegal characters to legal ones.

- User-defined names can contain A–Z, a–z, $, _, –, <, and >. A “/” is also valid, however; it is not recommended because it is used as a hierarchy separator.
- Names must contain at least one non-numeric character.
- Names cannot be more than 256 characters long.

The following FPGA resource names are reserved and should not be used to name nets or components.

- Components (Comps), Configurable Logic Blocks (CLBs), Input/Output Blocks (IOBs), Slices, basic elements (bels), clock buffers (BUFGs), tristate buffers (BUFTs), oscillators (OSC), CCLK, DP, GND, VCC, and RST
- CLB names such as AA, AB, SLICE_R1C2, SLICE_X1Y2, X1Y2, and R1C2
- Primitive names such as TD0, BSCAN, M0, M1, M2, or STARTUP
- Do not use pin names such as P1 and A4 for component names
- Do not use pad names such as PAD1 for component names

Refer to the language reference manual for Verilog or VHDL for language-specific naming restrictions. Xilinx® does not recommend using escape sequences for illegal characters. Also, if you plan to import schematics into your design, use the most restrictive character set.

## Matching File Names to Entity and Module Names

Xilinx® recommends the following practices in naming your HDL files.

- Ensure that the VHDL or Verilog source code file name matches the designated name of the entity (VHDL) or module (Verilog) specified in your design file. This is less confusing and generally makes it easier to create a script file for the compilation of your design.
- If your design contains more than one entity or module, each should be contained in a separate file with the appropriate file name.
- It is a good idea to use the same name as your top-level design file for your synthesis script file with either a .do, .scr, .script, or the appropriate default script file extension for your synthesis tool.

## Naming Identifiers, Types, and Packages

You can use long (256 characters maximum) identifier names with underscores and embedded punctuation in your code. Use meaningful names for signals and variables, such as CONTROL_REGISTER. Use meaningful names when defining VHDL types and packages as shown in the following examples.

```
type LOCATION_TYPE is ...;
package STRING_IO_PKG is
```

## Labeling Flow Control Constructs

You can use optional labels on flow control constructs to make the code structure more obvious, as shown in the following VHDL and Verilog examples. However, you should note that these labels are not translated to gate or register names in your implemented design. Flow control constructs can slow down simulations in some Verilog simulators.

- VHDL Example

```
-- D_REGISTER.VHD
-- Changing Latch into a D-Register

library IEEE;
use IEEE.std_logic_1164.all;

entity d_register is
  port (
       CLK, DATA: in STD_LOGIC;
       Q: out STD_LOGIC
       );
end d_register;
architecture BEHAV of d_register is
begin
  My_D_Reg: process (CLK, DATA)
  begin
    if (CLK’event and CLK=’1’) then
       Q <= DATA;
    end if;
  end process; --End My_D_Reg
end BEHAV;
```

- Verilog Example

```
/* Changing Latch into a D-Register
* D_REGISTER.V
*/

module d_register (CLK, DATA, Q);

input CLK;
input DATA;
output Q;

reg Q;

always @ (posedge CLK)
  begin: My_D_Reg
    Q <= DATA;
  end
endmodule
```

## Using Named and Positional Association

Xilinx® suggests that you always use named association to prevent incorrect connections for the ports of instantiated components. Do not combine positional and named association in the same statement as illustrated in the following examples.

- VHDL

  Incorrect

```
CLK_1: BUFGS port map (I=>CLOCK_IN,CLOCK_OUT);
```

  Correct

```
CLK_1: BUFGS port map(I=>CLOCK_IN,O=>CLOCK_OUT);
```

- Verilog

  Incorrect

  ```
  BUFGS CLK_1 (.I(CLOCK_IN), CLOCK_OUT);
  ```

  Correct

  ```
  BUFGS CLK_1 (.I(CLOCK_IN), .O(CLOCK_OUT));
  ```

## Passing Attributes

An attribute is attached to HDL objects in your design. You can pass attributes to HDL objects in two ways; you can predefine data that describes an object, or directly attach an attribute to an HDL object. Predefined attributes can be passed with a command file or constraints file in your synthesis tool, or you can place attributes directly in your HDL code. This section illustrates passing attributes in HDL code only. For information on passing attributes via the command file, please refer to your synthesis tool manual.

Most vendors adopt identical syntax for passing attributes in VHDL, but not in Verilog. The following examples illustrate the VHDL syntax.

### VHDL Attribute Examples

The following are examples of VHDL attributes.

- Attribute declaration:

  **attribute** *attribute_name* : *attribute_type;*

- Attribute use on a port or signal:

  **attribute** *attribute_name* **of** *object_name* : **signal is** *attribute_value*

  Example:

  ```
  library IEEE;
  use IEEE.std_logic_1164.all;
  entity d_register is
    port (
          CLK, DATA: in STD_LOGIC;
          Q: out STD_LOGIC);
    attribute FAST : string;
    attribute FAST of Q : signal is "true";
  end d_register;
  ```

- Attribute use on an instance:

  **attribute** *attribute_name* **of** *object_name* : **label is** *attribute_value*

  Example:

  ```
  architecture struct of spblkrams is
  attribute INIT_00: string;
  attribute INIT_00 of INST_RAMB4_S4: label is
  "1F1E1D1C1B1A191817161514131211100F0E0D0C0B0908706050403020100";
  begin
    INST_RAMB4_S4 : RAMB4_S4 port map (
        DI => DI(3 downto 0),
        EN => EN,
        WE => WE,
        RST => RST,
        CLK => CLK,
        ADDR => ADDR(9 downto 0),
        DO => DORAMB4_S4);
  ```

- Attribute use on a component:

  **attribute** *attribute_name* **of** *object_name* **: component is** *attribute_value*

  Example:

  ```
  architecture xilinx of tenths_ex is
  attribute black_box : boolean;
  component tenths
    port (
          CLOCK : in STD_LOGIC;
          CLK_EN : in STD_LOGIC;
          Q_OUT : out STD_LOGIC_VECTOR(9 downto 0)
          );
  end component;
  attribute black_box of tenths : component is true;
  begin
  ```

## Verilog Attribute Examples

The following are examples of attribute passing in Verilog. Note that attribute passing in Verilog is synthesis tool specific.

- Attribute use in FPGA Compiler II™™ syntax:

  //synopsys **attribute** *name value*

  Example:

  ```
  BUFG CLOCKB (.I(oscout), .O(clkint)); //synopsys
  attribute LOC "BR"
  ```

  or

  ```
  RAMB4_S4 U1 ( .WE(w), .EN(en), .RST(r), .CLK(ck)
  .ADDR(ad), .DI(di), .DO(do)); /* synopsys attribute INIT_00
  "AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBB" INIT_09
  "99999988888888887777777776666666" */
  ```

- Attribute use in LeonardoSpectrum™ syntax:

  //exemplar **attribute** *object_name attribute_name attribute_value*

  Examples:

  ```
  RAMB4_S8 U0 (.WE(WE), .EN(1’b1), .RST(1’b0),
  .CLK(CLK), .ADDR(ADDR), .DI(DIN), .DO(DOUT));
  //exemplar attribute U0 INIT_00
  1F1E1D1C1B1A191817161514131211100F0E0D0C0B0A0908070605040
  3020100
  ```

- Attribute use in Synplify™ syntax:

  // **synthesis** *directive*<br>
  //**synthesis** *attribute_name***=***value*

  or

  /* **synthesis** *directive */*<br>
  /* **synthesis** *attribute_name***=***value*>*/

  Examples:

  ```
  FDCE u2( .D (q1), .CE(ce), .C (clk), .CLR (rst),.Q (qo))
  /* synthesis rloc="r1c0.s0" */;
  ```

  or

```
module BUFG(I,O); // synthesis black_box
  input I;
  output O;
endmodule
```

## Understanding Synthesis Tools Naming Convention

Some net and logic names are preserved and some are altered by the synthesis tools during the synthesis process. This may result in a netlist that is hard to read or trace back to the original code.

This section discusses how different synthesis tools generate names from your VHDL / Verilog codes. This helps you corollate nets and component names appearing in the EDIF netlist. It also helps corollate nets and names during your after-synthesis design view of the VHDL / Verilog source.

***Note:*** The following naming conventions apply to inferred logic. The names of instantiated components and their connections, and port names are preserved during synthesis.

- FPGA Compiler II™ Naming Styles:

  Register instance: *outputsignal*_reg

  Output of register: *outputsignal*_reg

  Output of clock buffer: *signal*_BUFGed

  Output of tristate: *signal*_tri

  Port names: preserved

  Hierarchy notation: '_', e.g., *hier_1_hier_2*

  Other inferred component and net names are machine generated.

- LeonardoSpectrum™ Naming Styles:

  Register instance: reg_*outputsignal*

  Output of register: preserved, except if the output is also an external port of the design. In this case, it is *signal*_dup0

  Clock buffer / ibuf: *driversignal*_ibuf

  Output of clock buffer / ibuf: *driversignal*_int

  Tristate instance: tri_*outputsignal*

  Driver and output of tristate: preserved

  Hierarchy notation: '_'

  Other names are machine generated.

- Synplify™ Naming Styles:

  Register instance: output_signal

  Output of register: output_signal

  Clock buffer instance / ibuf: *portname*_ibuf

  Output of clock buffer: *clkname*_c

  Output / inout tristate instance: *outputsignal*_obuft or *outputsignal*_iobuf

  Internal tristate instance: un*n*_*signalname*_tb, when *n* is any number or *signalname*_tb

Output of tristate driving an output/inout : name of port

Output of internal tristate: *signalname*_tb_*number*

RAM instance and its output

- Dual Port RAM:

  ram instance: *memoryname_n*.I_*n*

  ram output : DPO->*memoryname_n*.rout_bus, SPO->*memory_name_n*.wout_bus

- Single Port RAM:

  ram instance: *memoryname*.I_*n*

  ram output: *memoryname*

- Single Port Block SelectRAM™:

  ram_instance: *memoryname*.I_*n*

  ram output: *memoryname*

- Dual Port Block SelectRAM™:

  ram_instance: *memory_name*.I_*n*

  ram output: *memoryname*[the output that is used]

Hierarchy delimiter is usually a ".", however when syn_hier="hard", the hierarchy delimiter in the edif is "/"

Other names are machine generated.

# Specifying Constants

Use constants in your design to substitute numbers to more meaningful names. The use of constants helps make a design more readable and portable.

## Using Constants to Specify OPCODE Functions (VHDL)

Do not use variables for constants in your code. Define constant numeric values in your code as constants and use them by name. This coding convention allows you to easily determine if several occurrences of the same literal value have the same meaning. In some simulators, using constants allows greater optimization. In the following code example, the OPCODE values are declared as constants, and the constant names refer to their function. This method produces readable code that may be easier to modify.

```
constant ZERO   : STD_LOGIC_VECTOR (1 downto 0):=“00”;
constant A_AND_B: STD_LOGIC_VECTOR (1 downto 0):=“01”;
constant A_OR_B : STD_LOGIC_VECTOR (1 downto 0):=“10”;
constant ONE    : STD_LOGIC_VECTOR (1 downto 0):=“11”;

process (OPCODE, A, B)
begin
  if (OPCODE = A_AND_B)then OP_OUT <= A and B;
    elsif (OPCODE = A_OR_B) then
        OP_OUT <= A or B;
    elsif (OPCODE = ONE) then
        OP_OUT <= ‘1’;
    else
        OP_OUT <= ‘0’;
  end if;
end process;
```

### Using Parameters to Specify OPCODE Functions (Verilog)

You can specify a constant value in Verilog using the parameter special data type, as shown in the following examples. The first example includes a definition of OPCODE constants as shown in the previous VHDL example. The second example shows how to use a parameter statement to define module bus widths.

- Example 1

```
//Using parameters for OPCODE functions
parameter ZERO = 2’b00;
parameter A_AND_B = 2’b01;
parameter A_OR_B = 2’b10;
parameter ONE = 2’b11;
always @ (OPCODE or A or B)
  begin
    if (OPCODE == ‘ZERO)
        OP_OUT = 1’b0;
    else if (OPCODE == ‘A_AND_B)
        OP_OUT=A&B;
    else if (OPCODE == ‘A_OR_B)
        OP_OUT = A|B;
    else
        OP_OUT = 1’b1;
  end
```

- Example 2

```
// Using a parameter for Bus Size
parameter BUS_SIZE = 8;

output [‘BUS_SIZE-1:0] OUT;
input  [‘BUS_SIZE-1:0] X,Y;
```

## Choosing Data Type (VHDL only)

Use the Std_logic (IEEE 1164) standards for hardware descriptions when coding your design. These standards are recommended for the following reasons.

- *Applies as a wide range of state values*—It has nine different values that represent most of the states found in digital circuits.

- *Automatically initializes to an unknown value*—Automatic initialization is important for HDL designs because it forces you to initialize your design to a known state, which is similar to what is required in a schematic design. Do not override this feature by initializing signals and variables to a known value when they are declared because the result may be a gate-level circuit that cannot be initialized to a known value.
- *Easily performs board-level simulation*—For example, if you use an integer type for ports for one circuit and standard logic for ports for another circuit, your design can be synthesized; however, you need to perform time-consuming type conversions for a board-level simulation.

The back-annotated netlist from Xilinx® implementation is in Std_logic. If you do not use Std_logic type to drive your top-level entity in the test bench, you cannot reuse your functional test bench for timing simulation. Some synthesis tools can create a wrapper for type conversion between the two top-level entities; however, this is not recommended by Xilinx®.

## Declaring Ports

Xilinx® recommends that you use the Std_logic package for all entity port declarations. This package makes it easier to integrate the synthesized netlist back into the design hierarchy without requiring conversion functions for the ports. The following is a VHDL example using the Std_logic package for port declarations.

```
Entity alu is
  port(
       A   : in STD_LOGIC_VECTOR(3 downto 0);
       B   : in STD_LOGIC_VECTOR(3 downto 0);
       CLK : in STD_LOGIC;
       C   : out STD_LOGIC_VECTOR(3 downto 0)
       );
end alu;
```

Since the downto convention for vectors is supported in a back-annotated netlist, the RTL and synthesized netlists should use the same convention if you are using the same test bench. This is necessary because of the loss of directionality when your design is synthesized to an EDIF netlist.

## Minimizing the Use of Ports Declared as Buffers

Do not use buffers when a signal is used internally and as an output port. In the following VHDL example, signal C is used internally and as an output port.

```
Entity alu is
  port(
       A   : in STD_LOGIC_VECTOR(3 downto 0);
       B   : in STD_LOGIC_VECTOR(3 downto 0);
       CLK : in STD_LOGIC;
       C   : buffer STD_LOGIC_VECTOR(3 downto 0) );
end alu;
architecture BEHAVIORAL of alu is
begin
  process begin
    if (CLK’event and CLK=’1’) then
       C <= UNSIGNED(A) + UNSIGNED(B) UNSIGNED(C);
    end if;
  end process;
end BEHAVIORAL;
```

Because signal C is used both internally and as an output port, every level of hierarchy in your design that connects to port C must be declared as a buffer. However, buffer types are not commonly used in VHDL designs because they can cause problems during synthesis. To reduce the amount of buffer coding in hierarchical designs, you can insert a dummy signal and declare port C as an output, as shown in the following VHDL example.

```
Entity alu is
  port(
        A   : in STD_LOGIC_VECTOR(3 downto 0);
        B   : in STD_LOGIC_VECTOR(3 downto 0);
        CLK : in STD_LOGIC;
        C   : out STD_LOGIC_VECTOR(3 downto 0)
        );
end alu;

architecture BEHAVIORAL of alu is
-- dummy signal
  signal C_INT : STD_LOGIC_VECTOR(3 downto 0);
  begin
    C <= C_INT;
    process begin
      if (CLK’event and CLK=’1’) then
          C_INT < =UNSIGNED(A) + UNSIGNED(B) + UNSIGNED(C_INT);
      end if;
    end process;
end BEHAVIORAL;
```

## Comparing Signals and Variables (VHDL only)

You can use signals and variables in your designs. Signals are similar to hardware and are not updated until the end of a process. Variables are immediately updated and, as a result, can affect the functionality of your design. Xilinx® recommends using signals for hardware descriptions; however, variables allow quick simulation.

The following VHDL examples show a synthesized design that uses signals and variables, respectively. These examples are shown implemented with gates in the “Gate Implementation of XOR_VAR” and “Gate Implementation of XOR_SIG” figures.

***Note:*** If you assign several values to a signal in one process, only the final value is used. When you assign a value to a variable, the assignment takes place immediately. A variable maintains its value until you specify a new value.

## Using Signals (VHDL)

```
-- XOR_SIG.VHD
Library IEEE;
use IEEE.std_logic_1164.all;
entity xor_sig is

  port (
        A, B, C: in  STD_LOGIC;
        X, Y: out STD_LOGIC
        );
end xor_sig;

architecture SIG_ARCH of xor_sig is
  signal D: STD_LOGIC;
  begin
    SIG:process (A,B,C)
    begin
      D <= A; -- ignored !!
      X <= C xor D;
      D <= B; -- overrides !!
      Y <= C xor D;
    end process;
end SIG_ARCH;
```

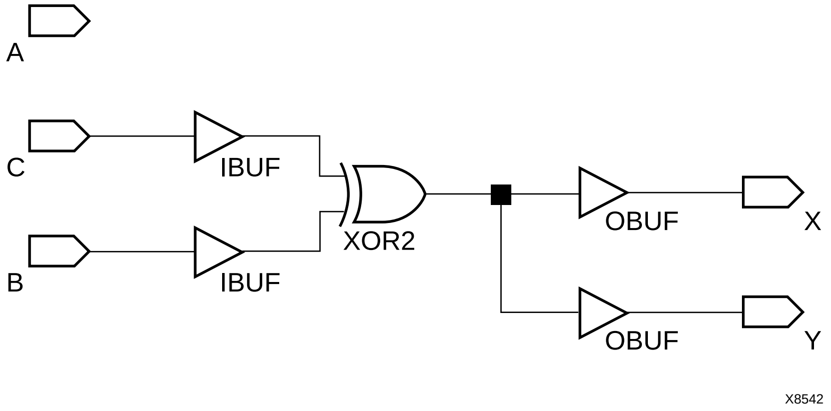


*Figure 3-1:* **Gate implementation of XOR_SIG**

### Using Variables (VHDL)

```
-- XOR_VAR.VHD

Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity xor_var is
  port (
      A, B, C: in  STD_LOGIC;
      X, Y:    out STD_LOGIC
      );
end xor_var;

architecture VAR_ARCH of xor_var is
begin
  VAR:process (A,B,C)
    variable D: STD_LOGIC;
    begin
      D := A;
      X <= C xor D;
      D := B;
      Y <= C xor D;
    end process;
end VAR_ARCH;
```

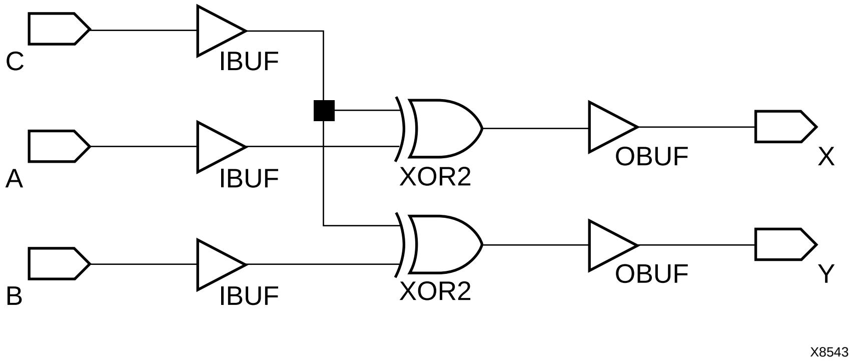


*Figure 3-2:* **Gate Implementation of XOR_VAR**

## Coding for Synthesis

VHDL and Verilog are hardware description and simulation languages that were not originally intended as inputs to synthesis. Therefore, many hardware description and simulation constructs are not supported by synthesis tools. In addition, the various synthesis tools use different subsets of VHDL and Verilog. VHDL and Verilog semantics are well defined for design simulation. The synthesis tools must adhere to these semantics to ensure that designs simulate the same way before and after synthesis. Follow the guidelines in the following sections to create code that simulates the same way before and after synthesis.

## Omit the Wait for XX ns Statement

Do not use the Wait for XX ns statement in your code. XX specifies the number of nanoseconds that must pass before a condition is executed. This statement does not synthesize to a component. In designs that include this statement, the functionality of the simulated design does not match the functionality of the synthesized design. VHDL and Verilog examples of the Wait for XX ns statement are as follows.

- VHDL

```
wait for XX ns;
```

- Verilog

```
    #XX;
```

## Omit the ...After XX ns or Delay Statement

Do not use the ...After XX ns statement in your VHDL code or the Delay assignment in your Verilog code. Examples of these statements are as follows.

- VHDL

```
(Q <=0 after XX ns)
```

- Verilog

```
assign #XX Q=0;
```

XX specifies the number of nanoseconds that must pass before a condition is executed. This statement is usually ignored by the synthesis tool. In this case, the functionality of the simulated design does not match the functionality of the synthesized design.

## Omit Initial Values

Do not assign signals and variables initial values because initial values are ignored by most synthesis tools. The functionality of the simulated design may not match the functionality of the synthesized design.

For example, do not use initialization statements like the following VHDL and Verilog statements.

- VHDL

```
signal sum : integer := 0;
```

- Verilog

```
initial sum = 1’b0;
```

## Order and Group Arithmetic Functions

The ordering and grouping of arithmetic functions can influence design performance. For example, the following two VHDL statements are not necessarily equivalent.

```
ADD <= A1 + A2 + A3 + A4;
ADD <= (A1 + A2) + (A3 + A4);
```

For Verilog, the following two statements are not necessarily equivalent.

```
ADD = A1 + A2 + A3 + A4;
ADD = (A1 + A2) + (A3 + A4);
```

The first statement cascades three adders in series. The second statement creates two adders in parallel: A1 + A2 and A3 + A4. In the second statement, the two additions are evaluated in parallel and the results are combined with a third adder. RTL simulation results are the same for both statements, however, the second statement results in a faster circuit after synthesis (depending on the bit width of the input signals).

Although the second statement generally results in a faster circuit, in some cases, you may want to use the first statement. For example, if the A4 signal reaches the adder later than the other signals, the first statement produces a faster implementation because the cascaded structure creates fewer logic levels for A4. This structure allows A4 to catch up to the other signals. In this case, A1 is the fastest signal followed by A2 and A3; A4 is the slowest signal.

Most synthesis tools can balance or restructure the arithmetic operator tree if timing constraints require it. However, Xilinx® recommends that you code your design for your selected structure.

## Comparing If Statement and Case Statement

The If statement generally produces priority-encoded logic and the Case statement generally creates balanced logic. An If statement can contain a set of different expressions while a Case statement is evaluated against a common controlling expression. In general, use the Case statement for complex decoding and use the If statement for speed critical paths.

Most current synthesis tools can determine if the if-elsif conditions are mutually exclusive, and will not create extra logic to build the priority tree. The following are points to consider when writing if statements.

- Make sure that all outputs are defined in all branches of an if statement. If not, it can create latches or long equations on the CE signal. A good way to prevent this is to have default values for all outputs before the if statements.
- Limiting the number of input signals into an if statement can reduce the number of logic levels. If there are a large number of input signals, see if some of them can be pre-decoded and registered before the if statement.
- Avoid bringing the dataflow into a complex if statement. Only control signals should be generated in complex if-else statements.

The following examples use an If construct in a 4–to–1 multiplexer design.

## 4–to–1 Multiplexer Design with If Construct

- VHDL Example

```
-- IF_EX.VHD
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity if_ex is
  port (
        SEL: in STD_LOGIC_VECTOR(1 downto 0);
        A,B,C,D: in STD_LOGIC;
        MUX_OUT: out STD_LOGIC);
end if_ex;

architecture BEHAV of if_ex is
begin

  IF_PRO: process (SEL,A,B,C,D)
    begin
      if (SEL=”00”) then MUX_OUT <= A;
      elsif (SEL=”01”) then
          MUX_OUT <= B;
      elsif (SEL=”10”) then
          MUX_OUT <= C;
      elsif (SEL=”11”) then
          MUX_OUT <= D;
      else
          MUX_OUT <= '0';
      end if;
    end process; --END IF_PRO
end BEHAV;
```

- Verilog Example

```
///////////////////////////////////////////////
// IF_EX.V                                   //
// Example of a If statement showing a       //
// mux created using priority encoded logic  //
// HDL Synthesis Design Guide for FPGAs      //
///////////////////////////////////////////////

module if_ex (A, B, C, D, SEL, MUX_OUT);

  input A, B, C, D;
  input [1:0] SEL;
  output MUX_OUT;
  reg MUX_OUT;

always @ (A or B or C or D or SEL)
  begin
    if (SEL == 2’b00)
        MUX_OUT = A;
    else if (SEL == 2’b01)
        MUX_OUT = B;
    else if (SEL == 2’b10)
        MUX_OUT = C;
    else if (SEL == 2’b11)
        MUX_OUT = D;
    else
        MUX_OUT = 0;
    end
endmodule
```

## 4–to–1 Multiplexer Design with Case Construct

The following VHDL and Verilog examples use a Case construct for the same multiplexer. Figure 3-3, page 62 shows the implementation of these designs. In these examples, the Case implementation requires only one Virtex™ slice while the If construct requires two slices in some synthesis tools. In this case, design the multiplexer using the Case construct because fewer resources are used and the delay path is shorter.

When writing case statements, make sure all outputs are defined in all branches.

- VHDL Example

```
-- CASE_EX.VHD
-- May 2001
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity case_ex is
  port (
        SEL : in STD_LOGIC_VECTOR(1 downto 0);
        A,B,C,D: in STD_LOGIC;
        MUX_OUT: out STD_LOGIC);
end case_ex;

architecture BEHAV of case_ex is
begin
  CASE_PRO: process (SEL,A,B,C,D)
  begin
      case SEL is
        when “00” => MUX_OUT <= A;
        when “01” => MUX_OUT <= B;
        when “10” => MUX_OUT <= C;
        when “11” => MUX_OUT <= D;
        when others => MUX_OUT <= '0';
      end case;
  end process; --End CASE_PRO
end BEHAV;
```

- Verilog Example

```
/////////////////////////////////////////////
// CASE_EX.V                              //
// Example of a Case statement showing   //
// A mux created using parallel logic    //
// HDL Synthesis Design Guide for FPGAs //
/////////////////////////////////////////////
module case_ex (A, B, C, D, SEL, MUX_OUT);
  input A, B, C, D;
  input [1:0] SEL;
  output MUX_OUT;
  reg MUX_OUT;

always @ (A or B or C or D or SEL)
  begin
    case (SEL)
      2’b00: MUX_OUT = A;
      2’b01: MUX_OUT = B;
      2’b10: MUX_OUT = C;
      2’b11: MUX_OUT = D;
      default: MUX_OUT = 0;
    endcase
  end
endmodule
```

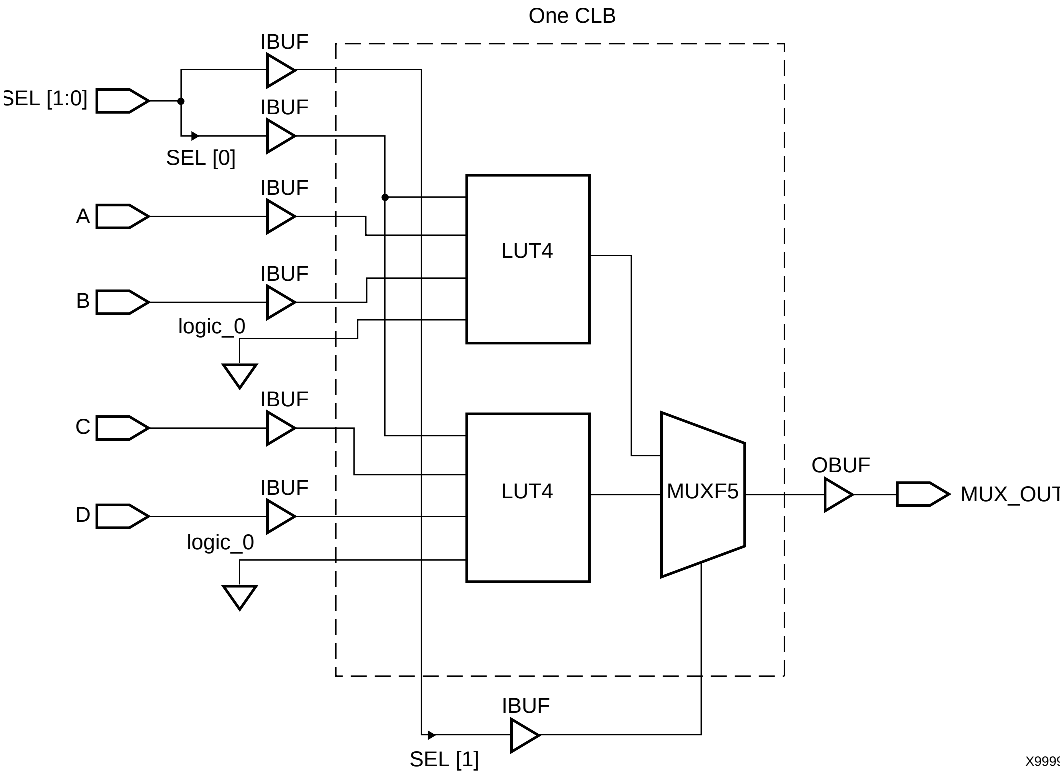


*Figure 3-3:* **Case_Ex Implementation**

## Implementing Latches and Registers

Synthesizers infer latches from incomplete conditional expressions, such as an If statement without an Else clause. This can be problematic for FPGA designs because not all FPGA devices have latches available in the CLBs. In addition, you may think that a register is created, and the synthesis tool actually created a latch. The Spartan-II™, Spartan-3™ and Virtex™, Virtex-E™, Virtex-II™, Virtex-II Pro™ and Virtex-II Pro X™ FPGAs do have registers that can be configured to act as latches. For these devices, synthesizers infer a dedicated latch from incomplete conditional expressions.

### D Latch Inference

- VHDL Example

```
-- D_LATCH.VHD

library IEEE;
use IEEE.std_logic_1164.all;

entity d_latch is
  port (
       GATE, DATA: in STD_LOGIC;
       Q: out STD_LOGIC);
end d_latch;

architecture BEHAV of d_latch is
begin
  LATCH: process (GATE, DATA)
  begin
    if (GATE = '1') then
       Q <= DATA;
    end if;
end process; -- end LATCH

end BEHAV;
```

- Verilog Example

```
/* Transparent High Latch
 * D_LATCH.V
*/

module d_latch (GATE, DATA, Q);

  input GATE;
  input DATA;
  output Q;
  reg Q;

always @ (GATE or DATA)
  begin
    if (GATE == 1'b1)
       Q <= DATA;
  end    // End Latch

endmodule
```

## Converting D Latch to D Register

If your intention is to not infer a latch, but rather to infer a D register, then the following code is the latch code example, modified to infer a D register.

- VHDL Example

```
-- D_REGISTER.VHD
-- Changing Latch into a D-Register

library IEEE;
use IEEE.std_logic_1164.all;

entity d_register is

  port (
       CLK, DATA: in STD_LOGIC;
       Q: out STD_LOGIC
       );
end d_register;

architecture BEHAV of d_register is
begin

  MY_D_REG: process (CLK, DATA)
  begin
    if (CLK’event and CLK=’1’) then
       Q <= DATA;
    end if;
  end process;    --End MY_D_REG
end BEHAV;
```

- Verilog Example

```
/* Changing Latch into a D-Register
 * D_REGISTER.V
*/

module d_register (CLK, DATA, Q);

  input CLK;
  input DATA;
  output Q;
  reg Q;
  always @ (posedge CLK)
    begin: My_D_Reg
      Q <= DATA;
    end
endmodule
```

With some synthesis tools you can determine the number of latches that are implemented in your design. Check the manuals that came with your software for information on determining the number of latches in your design.

You should convert all If statements without corresponding Else statements and without a clock edge to registers. Use the recommended register coding styles in the synthesis tool documentation to complete this conversion.

## Resource Sharing

Resource sharing is an optimization technique that uses a single functional block (such as an adder or comparator) to implement several operators in the HDL code. Use resource sharing to improve design performance by reducing the gate count and the routing congestion. If you do not use resource sharing, each HDL operation is built with separate circuitry. However, you may want to disable resource sharing for speed critical paths in your design.

The following operators can be shared either with instances of the same operator or with an operator on the same line.

```
*
+ –
> >= < <=
```

For example, a + operator can be shared with instances of other + operators or with – operators. A * operator can be shared only with other * operators.

You can implement arithmetic functions (+, –, magnitude comparators) with gates or with your synthesis tool's module library. The library functions use modules that take advantage of the carry logic in Spartan-II™, Spartan-3™, Virtex™ family, and Virtex-II Pro™ family CLBs / slices. Carry logic and its dedicated routing increase the speed of arithmetic functions that are larger than 4-bits. To increase speed, use the module library if your design contains arithmetic functions that are larger than 4-bits or if your design contains only one arithmetic function. Resource sharing of the module library automatically occurs in most synthesis tools if the arithmetic functions are in the same process.

Resource sharing adds additional logic levels to multiplex the inputs to implement more than one function. Therefore, you may not want to use it for arithmetic functions that are part of your design's time critical path.

Since resource sharing allows you to reduce the number of design resources, the device area required for your design is also decreased. The area that is used for a shared resource depends on the type and bit width of the shared operation. You should create a shared resource to accommodate the largest bit width and to perform all operations.

If you use resource sharing in your designs, you may want to use multiplexers to transfer values from different sources to a common resource input. In designs that have shared operations with the same output target, the number of multiplexers is reduced as illustrated in the following VHDL and Verilog examples. The HDL example is shown implemented with gates in Figure 3-4.

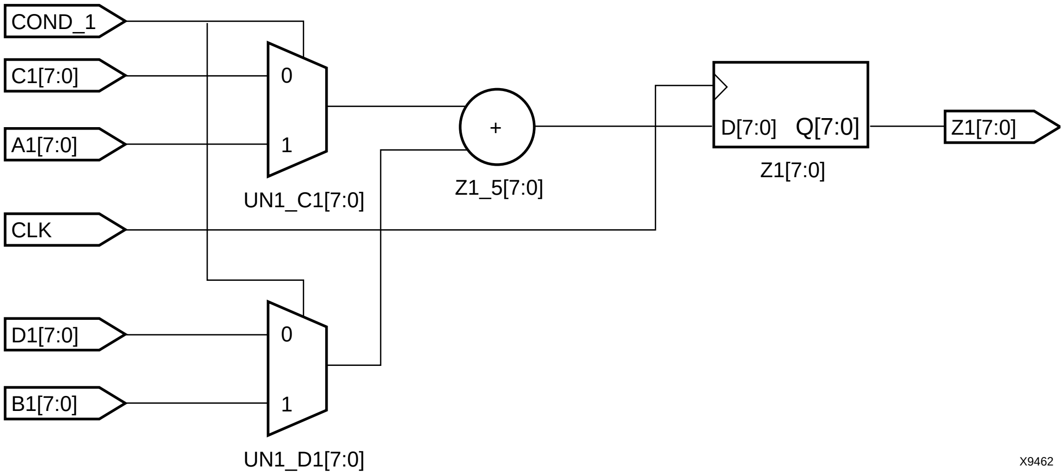


*Figure 3-4:* **Implementation of Resource Sharing**

- VHDL Example

```
-- RES_SHARING.VHD

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

entity res_sharing is
  port (
        A1,B1,C1,D1 : in STD_LOGIC_VECTOR (7 downto 0);
        COND_1 : in STD_LOGIC;
        Z1 : out STD_LOGIC_VECTOR (7 downto 0));
end res_sharing;

architecture BEHAV of res_sharing is
begin
  P1: process (A1,B1,C1,D1,COND_1)
  begin
    if (COND_1='1') then
        Z1 <= A1 + B1;
    else
        Z1 <= C1 + D1;
    end if;
  end process; -- end P1

end BEHAV;
```

- Verilog Example

```
/* Resource Sharing Example
 * RES_SHARING.V
*/

module res_sharing (A1, B1, C1, D1, COND_1, Z1);

input COND_1;
input [7:0] A1, B1, C1, D1;
output [7:0] Z1;
reg [7:0] Z1;

always @(A1 or B1 or C1 or D1 or COND_1)
  begin
    if (COND_1)
        Z1 <= A1 + B1;
    else
        Z1 <= C1 + D1;
  end
endmodule
```

If you disable resource sharing or if you code the design with the adders in separate processes, the design is implemented using two separate modules as shown in Figure 3-5

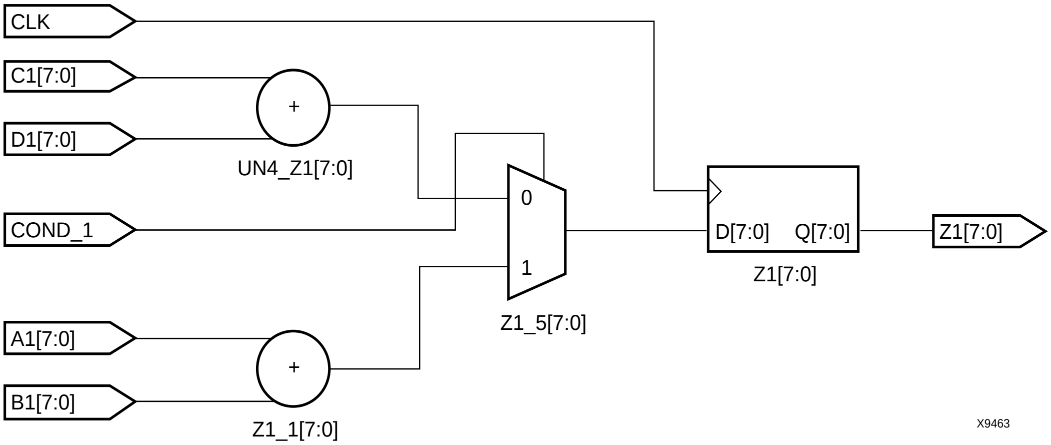


*Figure 3-5:* **Implementation without Resource Sharing**

***Note:*** Refer to the appropriate reference manual for more information on resource sharing.

## Reducing Gate Count

Use the generated module components to reduce the number of gates in your designs. The module generation algorithms use Xilinx® carry logic to reduce function generator logic and improve routing and speed performance. Further gate reduction can occur with synthesis tools that recognize the use of constants with the modules.

You can reduce the number of gates further by mapping your design onto dedicated logic blocks such as Block RAM. This also reduces the amount of distributed logic.

## Using Preset Pin or Clear Pin

Xilinx® FPGAs consist of CLBs that contain function generators and flip-flops. Spartan-II™, Spartan-3™ , Virtex™, Virtex-E™, Virtex-II™, Virtex-II Pro™ and Virtex-II Pro X™ registers can be configured to have either or both preset and clear pins.

### Register Inference

The following VHDL and Verilog designs show how to describe a register with a clock enable and either an asynchronous preset or a clear.

- VHDL Example

```
-- FF_EXAMPLE.VHD
-- Example of Implementing Registers

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity ff_example is
  port ( RESET, SET, CLOCK, ENABLE: in STD_LOGIC;
     D_IN: in STD_LOGIC_VECTOR (7 downto 0);
     A_Q_OUT: out STD_LOGIC_VECTOR (7 downto 0);
     B_Q_OUT: out STD_LOGIC_VECTOR (7 downto 0);
     C_Q_OUT: out STD_LOGIC_VECTOR (7 downto 0);
     D_Q_OUT: out STD_LOGIC_VECTOR (7 downto 0);
     E_Q_OUT: out STD_LOGIC_VECTOR (7 downto 0));

end ff_example;

architecture BEHAV of ff_example is
begin
   -- D flip-flop
   FF: process (CLOCK)
   begin
      if (CLOCK’event and CLOCK=’1’) then
         A_Q_OUT <= D_IN;
      end if;
   end process; -- End FF
   -- Flip-flop with asynchronous reset
   FF_ASYNC_RESET: process (RESET, CLOCK)
   begin
      if (RESET = ’1’) then
         B_Q_OUT <= “00000000”;
      elsif (CLOCK'event and CLOCK='1') then
         B_Q_OUT <= D_IN;
      end if;
   end process; -- End FF_ASYNC_RESET
   -- Flip-flop with asynchronous set
   FF_ASYNC_SET: process (SET, CLOCK)
   begin
      if (SET = '1') then
         C_Q_OUT <= “11111111”;
      elsif (CLOCK'event and CLOCK = '1') then
         C_Q_OUT <= D_IN;
      end if;
   end process; -- End FF_ASYNC_SET
```

```
-- Flip-flop with asynchronous reset
-- and clock enable
    FF_CLOCK_ENABLE: process (ENABLE, RESET, CLOCK)
    begin
        if (RESET = ’1’) then
            D_Q_OUT <= “00000000”;
        elsif (CLOCK'event and CLOCK='1') then
            if (ENABLE = '1') then
              D_Q_OUT <= D_IN;
            end if;
        end if;
    end process; -- End FF_CLOCK_ENABLE

-- Flip-flop with asynchronous reset
-- asynchronous set and clock enable
    FF_ASR_CLOCK_ENABLE: process (ENABLE, RESET, SET, CLOCK)
    begin
        if (RESET = '1') then
            E_Q_OUT <= "00000000";
        elsif (SET = '1') then
            E_Q_OUT <= "11111111";
        elsif (CLOCK'event and CLOCK='1') then
            if (ENABLE = '1') then
                E_Q_OUT <= D_IN;
            end if;
        end if;
    end process; -- End FF_ASR_CLOCK_ENABLE

end BEHAV;
```

- Verilog Example

```
-- FF_EXAMPLE.V
-- Example of Implementing Registers

module ff_example( RESET, SET, CLOCK, ENABLE;D_IN;
A_Q_OUT; B_Q_OUT; C_Q_OUT; D_Q_OUT; E_Q_OUT);

input RESET;
input SET;
input CLOCK;
input ENABLE;
input [7:0] D_IN;
output [7:0] A_Q_OUT;
output [7:0] B_Q_OUT;
output [7:0] C_Q_OUT;
output [7:0] D_Q_OUT;
output [7:0] E_Q_OUT;

    // D flip-flop
    always @(posedge CLOCK)
    begin
      A_Q_OUT <= D_IN;
    end      // End FF
    // Flip-flop with asynchronous reset
    always @(posedge CLOCK || posedge RESET)
      begin
        if (RESET == 1’b1)
            B_Q_OUT <= “00000000”;
        else if (CLOCK == 1'b1)
            B_Q_OUT <= D_IN;
      end    // End Flip-flop with asynchronous reset

    // Flip-flop with asynchronous set
    always @(posedge CLOCK || posedge SET)
      begin
        if (SET == 1'b1)
            C_Q_OUT <= “11111111”;
        else if (CLOCK == 1'b1)
            C_Q_OUT <= D_IN;
      end    // End Flip-flop with asynchronous set

// Flip-flop with asynchronous reset
// and clock enable
    always @(posedge CLOCK || posedge RESET)
      begin
        if (RESET == 1'b1)
            D_Q_OUT <= “00000000”;
        else if (CLOCK == 1'b1)
          begin
            if (ENABLE == 1'b1)
                D_Q_OUT <= D_IN;
          end
      end    // End Flip-flop with asynchronous reset and clock enable

// Flip-flop with asynchronous reset
// asynchronous set and clock enable
    always @(posedge CLOCK || posedge RESET || posedge SET)
    begin
      if (RESET == 1'b1)
          E_Q_OUT <= "00000000";
      else if (SET == 1'b1)
          E_Q_OUT <= "11111111";
      else if (CLOCK == 1'b1)
          begin
            if (ENABLE == 1'b1)
                E_Q_OUT <= D_IN;
          end
    end    // End Flip-flop with asynchronous reset
           // asynchronous set and clock enable
end module
```

## Using Clock Enable Pin Instead of Gated Clocks

Use the CLB clock enable pin instead of gated clocks in your designs. Gated clocks can introduce glitches, increased clock delay, clock skew, and other undesirable effects. The first two examples in this section (VHDL and Verilog) illustrate a design that uses a gated clock. Figure 3-6 shows this design implemented with gates. Following these examples are VHDL and Verilog designs that show how you can modify the gated clock design to use the clock enable pin of the CLB. Figure 3-7 shows this design implemented with gates.

- VHDL Example

```
-------------------------------------------
-- GATE_CLOCK.VHD Version 1.1            --
-- Illustrates clock buffer control      --
-- Better implementation is to use       --
-- clock enable rather than gated clock --
-------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity gate_clock is
   port (IN1,IN2,DATA,CLK,LOAD: in STD_LOGIC;
         OUT1: out STD_LOGIC);
end gate_clock;

architecture BEHAVIORAL of gate_clock is
signal GATECLK: STD_LOGIC;

begin
GATECLK <= (IN1 and IN2 and CLK);
   GATE_PR: process (GATECLK,DATA,LOAD)
   begin
     if (GATECLK’event and GATECLK=’1’) then
       if (LOAD = ’1’) then
         OUT1 <= DATA;
       end if;
     end if;
   end process; -- End GATE_PR
end BEHAVIORAL;
```

- Verilog Example

```
//////////////////////////////////////////
// GATE_CLOCK.V Version 1.1             //
// Gated Clock Example                  //
// Better implementation to use clock   //
// enables than gating the clock        //
//////////////////////////////////////////

module gate_clock(IN1, IN2, DATA, CLK,LOAD,OUT1);
input IN1;
input IN2;
input DATA;
input CLK;
input LOAD;
output OUT1;
reg OUT1;
wire GATECLK;
assign GATECLK = (IN1 & IN2 & CLK);

always @(posedge GATECLK)
  begin
    if (LOAD == 1’b1)
        OUT1 <= DATA;
  end

endmodule
```

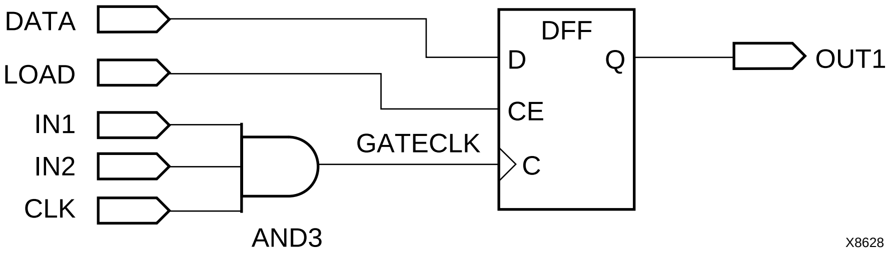


*Figure 3-6:* **Implementation of Gated Clock**

- VHDL Example

```
-- CLOCK_ENABLE.VHD

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity clock_enable is
    port (IN1,IN2,DATA,CLOCK,LOAD: in STD_LOGIC;
           DOUT: out STD_LOGIC);
end clock_enable;

architecture BEHAV of clock_enable is
signal ENABLE: STD_LOGIC;
begin

    ENABLE <= IN1 and IN2 and LOAD;

    EN_PR: process (ENABLE,DATA,CLOCK)
    begin
      if (CLOCK’event and CLOCK=’1’) then
        if (ENABLE = ’1’) then
          DOUT <= DATA;
        end if;
      end if;
    end process; -- End EN_PR

end BEHAV;
```

- Verilog Example

```
/* Clock enable example
 * CLOCK_ENABLE.V
 * May 2001
*/

module clock_enable (IN1, IN2, DATA, CLK, LOAD, DOUT);

  input IN1, IN2, DATA;
  input CLK, LOAD;
  output DOUT;

  wire ENABLE;
  reg DOUT;

  assign ENABLE = IN1 & IN2 & LOAD;

    always @(posedge CLK)
    begin
      if (ENABLE)
        DOUT <= DATA;
    end

endmodule
```

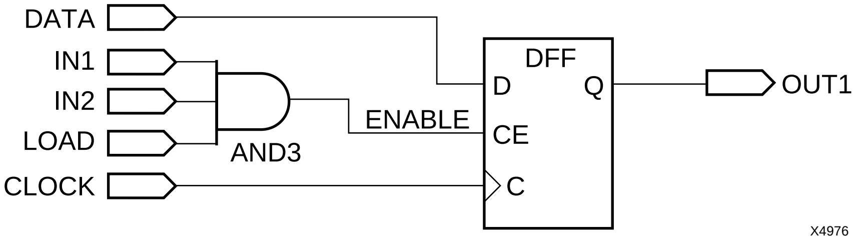


*Figure 3-7:* **Implementation of Clock Enable**

# Chapter 4

# Architecture Specific Coding Styles for Spartan™-II/-3,Virtex™/-E/-II/-II Pro/ -II Pro X

This chapter includes coding techniques to help you improve synthesis results. It includes the following sections.

## Introduction

This chapter highlights the features and synthesis techniques in designing with Xilinx® Virtex™, Virtex-E™, Virtex-II™, Virtex-II Pro™, Virtex-II Pro X™, Spartan™-II and Spartan-3™ FPGAs. Virtex™, Virtex-E™ and Spartan-II™ devices share many architectural similarities. Virtex-II™, Virtex-II Pro™, Virtex-II Pro X™ and Spartan-3™ provide an architecture that is substantially different from Virtex™, Virtex-E™ and Spartan-II™; however, many of the synthesis design techniques apply the same way to all these devices. Unless otherwise stated, the features and examples in this chapter apply to all Virtex™, Virtex-E™, Virtex-II™, Virtex-II Pro™, Virtex-II Pro X™, Spartan-II™ and Spartan-3™ devices. For details specific to Virtex-II Pro™ and Virtex-II Pro X™, see the *Virtex-II Pro Platform FPGA User Guide*.

This chapter covers the following FPGA HDL coding features.

- Advanced clock management
- On-chip RAM and ROM
- IEEE 1149.1 — compatible boundary scan logic support
- Flexible I/O with Adjustable Slew-rate Control and Pull-up/Pull-down Resistors
- Various drive strength
- Various I/O standards
- Dedicated high-speed carry-propagation circuit

You can use these device characteristics to improve resource utilization and enhance the speed of critical paths in your HDL designs. The examples in this chapter are provided to help you incorporate these system features into your HDL designs.

# Instantiating Components

Xilinx® provides a set of libraries that your Synthesis tool can infer from your HDL code description. However, architecture specific and customized components must be explicitly instantiated as components in your design.

## Instantiating FPGA Primitives

Architecture specific components that are built in to the implementation software's library are available for instantiation without the need for specifying a definition. These components are marked as *primitive* in the *Libraries Guide*. Components marked as *macro* in the *Libraries Guide* are not built into the implementation software's library so they cannot be instantiated. The macro components in the *Libraries Guide* define the schematic symbols. When macros are used, the schematic tool decomposes the macros into their primitive elements when the schematic tool writes out the netlist.

FPGA primitives can be instantiated in VHDL and Verilog.

- VHDL example (declaring component and port map)

```
library IEEE;
use IEEE.std_logic_1164.all;
-- Add the following two lines if using Synplify:
-- library virtex;
-- use virtex.components.all;
entity flops is port(
       di  : in std_logic;
       ce  : in std_logic;
       clk : in std_logic;
       qo  : out std_logic;
       rst : in std_logic);
end flops;
-- remove the following component declaration
-- if using Synplify
```

```
architecture inst of flops is
component FDCE port(
      D   : in std_logic;
      CE  : in std_logic;
      C   : in std_logic;
      CLR : in std_logic;
      Q   : out std_logic);
end component;

begin
U0 : FDCE port map(
      D   => di,
      CE  => ce,
      C   => clk,
      CLR => rst,
      Q   => qo);
end inst;
```

***Note:*** To use this example in Synplify™, you need to add the Xilinx® primitive library and remove the component declarations as noted above.

The Virtex™ library contains primitives of Virtex™ and Spartan-II™ architectures. Replace ‘virtex’ with the appropriate device family if you are targeting other Xilinx® FPGA architectures.

If you are designing with a Virtex-E™ device, use the **`virtexe`** library. If you are designing with a Virtex-II™, Virtex-II Pro™, Virtex-II Pro X™ and Spartan-3™ device, use the **`virtex2`** library.

- Verilog example

```
module flops (d1, ce, clk, q1, rst);
input d1;
input ce;
input clk;
output q1;
input rst;

FDCE u1 (
      .D (d1),
      .CE (ce),
      .C (clk),
      .CLR (rst),
      .Q (q1));
endmodule
```

## Instantiating CORE Generator™ Modules

The CORE Generator™ allows you to generate complex ready-to-use functions such as FIFO, Filter, Divider, RAM, and ROM. CORE Generator™ generates an EDIF netlist to describe the functionality and a component instantiation template for HDL instantiation. For more information on the use and functions created by the CORE Generator™, see the *CORE Generator Guide*.

In VHDL, you can declare the component and port map as shown in the *“Instantiating FPGA Primitives”* section above. Synthesis tools assume a black box for components that do not have a VHDL functional description.

In Verilog, an empty module must be declared to get port directionality. Synthesis tools assume a black box for components that do not have a Verilog functional description.

Example of Black Box Directive and Empty Module Declaration.

```
module r256x16s (addr, di, clk, we, en, rst, do);
input [7:0] addr;
input [15:0] di;
input clk;
input we;
input en;
input rst;
output [15:0] do;
endmodule

module top (addrp, dip, clkp, wep, enp, rstp, dop);
input [7:0] addrp;
input [15:0] dip;
input clkp;
input wep;
input enp;
input rstp;
output [15:0] dop;
r256x16s U0(
       .addr(addrp), .di(dip),
       .clk(clkp), .we(wep),
       .en(enp), .rst(rstp),
       .do(dop));
endmodule
```

## Using Boundary Scan (JTAG 1149.1)

Virtex™, Virtex-E™, Virtex- II™, Virtex-II Pro™, Virtex-II Pro X™, Spartan-II™ and Spartan-3™ FPGAs contain boundary scan facilities that are compatible with IEEE Standard 1149.1.

You can access the built-in boundary scan logic between power-up and the start of configuration.

In a configured Virtex™, Virtex-E™, Virtex- II™, Virtex-II Pro™, Virtex-II Pro X™, Spartan-II™ and Spartan-3™ device, basic boundary scan operations are always available. BSCAN_VIRTEX, BSCAN_VIRTEX2 and BSCAN_SPARTAN2 are instantiated only if you want to create internal boundary scan chains in a Virtex™, Virtex-E™, Virtex-II™, Virtex-II Pro™ or Spartan-II™ device.

For specific information on boundary scan for an architecture, refer to the *Libraries Guide* and the product Data Sheets. For information on configuration and readback of Virtex™, Virtex-E™ and Spartan-II™ FPGAs, see the XAPP139 Application Note.

## Using Global Clock Buffers

For designs with global signals, use global clock buffers to take advantage of the low-skew, high-drive capabilities of the dedicated global buffer tree of the target device. Your synthesis tool automatically inserts a clock buffer whenever an input signal drives a clock signal or whenever an internal clock signal reaches a certain fanout. The Xilinx® implementation software automatically selects the clock buffer that is appropriate for your specified design architecture.

Some synthesis tools also limit global buffer insertions to match the number of buffers available on the device. Refer to your synthesis tool documentation for detailed information.

You can instantiate the clock buffers if your design requires a special architecture-specific buffer or if you want to specify how the clock buffer resources should be allocated.

Table 4-1summarizes global buffer (BUFG) resources in Virtex™, Virtex-E™, Virtex-II™, Virtex-II Pro™, Virtex-II Pro X™, Spartan-II™ and Spartan-3™ devices.

*Table 4-1:* **Global Buffer Resources**

| Buffer Type | Virtex™ | Virtex-E™ | Virtex™-II/II Pro/II Pro X | Spartan-II™ | Spartan-3™ |
|---|---|---|---|---|---|
| BUFG | 4 | 4 | Implemented as BUFGMUX | 4 | Implemented as BUFGMUX |
| BUFGMUX | N/A | N/A | 16 | N/A | 8 |

Virtex™/E/II/II Pro/II Pro X and Spartan™-II/3 devices include two tiers of global routing resources referred to as primary global and secondary local clock routing resources.

***Note:*** In Virtex™-II/II Pro/II Pro X and Spartan-3™, BUFG is available for instantiation, but is implemented with BUFGMUX.

- The primary global routing resources are dedicated global nets with dedicated input pins that are designed to distribute high-fanout clock signals with minimal skew. Each global clock net can drive all CLB, IOB, and Block SelectRAM™ clock pins. The primary global nets may only be driven by the global buffers (BUFG), one for each global net. There are four primary global nets in Virtex™, Virtex-E™ and Spartan-II™. There are sixteen in Virtex-II™ and Virtex-II Pro™.
- The secondary local clock routing resources consist of backbone lines or longlines. These secondary resources are more flexible than the primary resources since they are not restricted to routing clock signal only. These backbone lines are accessed differently between Virtex™/E/Spartan-II™ and Virtex™-II/II Pro/II Pro X, Spartan-3™ devices as follows:
  - In Virtex™, Virtex-E™ and Spartan-II™ devices, there are 12 longlines across the top of the chip and 12 across the bottom. From these lines, up to 12 unique signals per column can be distributed via the 12 longlines in the column. To use this, you must specify the USELOWSKEWLINES constraint in the UCF file. For more information on the USELOWSKEWLINES constraint syntax, refer to the *Constraints Guide.*
  - In Virtex™-II, Virtex-II Pro™ and Virtex-II Pro X™ and Spartan-3™, longlines resources are more abundant. There are many ways in which the secondary clocks or high fanout signals can be routed using a pattern of resources that result in low skew. The Xilinx® Implementation tools automatically use these resources based on various constraints in your design. Additionally, the USELOWSKEWLINES constraint can be applied to access this routing resource.

## Inserting Clock Buffers

Many synthesis tools automatically insert a global buffer (BUFG) when an input port drives a register’s clock pin or when an internal clock signal reaches a certain fanout. A

BUFGP (an IBUFG-BUFG connection) is inserted for the external clock whereas a BUFG is inserted for an internal clock. Most synthesis tools also allow you to control BUFG insertions manually if you have more clock pins than the available BUFGs resources.

FPGA Compiler II™ infers up to four clock buffers for pure clock nets. FPGA Compiler II™ does not infer a BUFG on a clock line that only drives one flip-flop.You can also instantiate clock buffers or assign them via the Express Constraints Editor.

***Note:*** Synthesis tools currently insert simple clock buffers, BUFGs, for all Virtex™/E/II/II Pro/ II Pro X and Spartan™-II/3 designs. For Virtex™-II/II Pro/II Pro X and Spartan-3™, some tools provide an attribute to use BUFGMUX as an enabled clock buffer. To use BUFGMUX as a real clock multiplexer in Virtex™-II/II Pro/II Pro X and Spartan-3™, it must be instantiated.

LeonardoSpectrum™ forces clock signals to global buffers when the resources are available. The best way to control unnecessary BUFG insertions is to turn off global buffer insertion, then use the BUFFER_SIG attribute to push BUFGs onto the desired signals. By doing this you do not have to instantiate any BUFG components. As long as you use *chip* options to optimize the IBUFs, they are auto-inserted for the input.

The following is a syntax example of the BUFFER_SIG attribute.

```
set_attribute -port clk1 -name buffer_sig -value BUFG
set_attribute -port clk2 -name buffer_sig -value BUFG
```

Synplify™ assigns a BUFG to any input signal that directly drives a clock. The maximum number of global buffers is defined as 4. Auto-insertion of the BUFG for internal clocks occurs with a fanout threshold of 16 loads. To turn off automatic clock buffers insertion, use the syn_noclockbuf attribute. This attribute can be applied to the entire module/architecture or a specific signal. To change the maximum number of global buffer insertion, set an attribute in the SDC file as follows.

```
define_global_attribute xc_global buffers (8)
```

XST assigns a BUFG to any input signal that directly drives a clock. The default number of global buffers for the Virtex™, Virtex-E™ and Spartan-II™ device is 4. The default number of global buffers for the Virtex-II™, Virtex-II Pro™, Virtex-II Pro X™ and Spartan-3™ device is 8. The number of BUFGs used for a design can be modified by the XST option *bufg*. You can use the *bufg* option by inserting it in the HDL code, including it in the XST constraints file or invoking it with a command line switch.

Refer to your synthesis tool documentation for detailed syntax information.

## Instantiating Global Clock Buffers

You can instantiate global buffers in your code as described in this section.

### Instantiating Buffers Driven from a Port

You can instantiate global buffers and connect them to high-fanout ports in your code rather than inferring them from a synthesis tool script. If you do instantiate global buffers, verify that the Pad parameter is not specified for the buffer.

In Virtex™/E/II and Spartan-II™ designs, synthesis tools insert BUFGP for clock signals which access a dedicated clock pin. To have a regular input pin to a clock buffer connection, you must use an IBUF-BUFG connection. This is done by instantiating BUFG after disabling global buffer insertion.

## Instantiating Buffers Driven from Internal Logic

Some synthesis tools require you to instantiate a global buffer in your code to use the dedicated routing resource if a high-fanout signal is sourced from internal flip-flops or logic (such as a clock divider or multiplexed clock), or if a clock is driven from a non-dedicated I/O pin. If using Virtex™/E or Spartan-II™ devices, the following VHDL and Verilog examples instantiate a BUFG for an internal multiplexed clock circuit.

***Note:*** Synplify™ infers a global buffer for a signal that has 16 or greater fanouts.

- VHDL example

```
----------------------------------------------
-- CLOCK_MUX_BUFG.VHD Version 1.1           --
-- This is an example of an instantiation of --
-- global buffer (BUFG) from an internally   --
-- driven signal, a multiplexed clock.       --
----------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
entity clock_mux is
  port (
        DATA, SEL : in STD_LOGIC;
        SLOW_CLOCK, FAST_CLOCK : in STD_LOGIC;
        DOUT : out STD_LOGIC
        );
end clock_mux;
architecture XILINX of clock_mux is

signal CLOCK : STD_LOGIC;
signal CLOCK_GBUF : STD_LOGIC;
component BUFG
  port (
        I : in STD_LOGIC;
        O : out STD_LOGIC
        );
end component;
begin
Clock_MUX: process (SEL, FAST_CLOCK, SLOW_CLOCK)
    begin
      if (SEL = ’1’) then
          CLOCK <= FAST_CLOCK;
      else
          CLOCK <= SLOW_CLOCK;
      end if;
    end process;

GBUF_FOR_MUX_CLOCK: BUFG
  port map (
        I => CLOCK,
        O => CLOCK_GBUF
        );

Data_Path: process (CLOCK_GBUF)
  begin
    if (CLOCK_GBUF’event and CLOCK_GBUF=’1’)then
        DOUT <= DATA;
    end if;
  end process;
end XILINX;
```

- Verilog example

```
////////////////////////////////////////////////
// CLOCK_MUX_BUFG.V Version 1.1               //
// This is an example of an instantiation of  //
// global buffer (BUFG) from an internally    //
// driven signal, a multiplied clock.         //
////////////////////////////////////////////////
module clock_mux(DATA,SEL,SLOW_CLOCK,FAST_CLOCK,
                 DOUT);
   input  DATA, SEL;
   input  SLOW_CLOCK, FAST_CLOCK;
   output DOUT;
   reg    CLOCK;
   wire   CLOCK_GBUF;
   reg DOUT;
   always @ (SEL or FAST_CLOCK or SLOW_CLOCK)
   begin
     if (SEL == 1'b1)
        CLOCK <= FAST_CLOCK;
     else
        CLOCK <= SLOW_CLOCK;
   end
   BUFG GBUF_FOR_MUX_CLOCK (.O(CLOCK_GBUF),.I(CLOCK));
   always @ (posedge CLOCK_GBUF)
       DOUT <= DATA;
endmodule
```

If using a Virtex™-II / II Pro / II Pro X or Spartan-3™ device a BUFGMUX can be used to multiplex between clocks. The above examples are rewritten for Virtex™-II / II Pro / II Pro X or Spartan-3™:

- VHDL example

```
--------------------------------------------------
-- CLOCK_MUX_BUFG.VHD Version 1.2               --
-- This is an example of an instantiation of    --
-- a multiplexing global buffer (BUFGMUX)       --
-- from an internally driven signal             --
--------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity clock_mux is
  port (DATA, SEL              : in std_logic;
       SLOW_CLOCK, FAST_CLOCK : in std_logic;
       DOUT                   : out std_logic);
end clock_mux;

architecture XILINX of clock_mux is
  signal CLOCK_GBUF : std_logic;
  component BUFGMUX
   port (
       I0 : in std_logic;
       I1 : in std_logic;
       S  : in std_logic;
       O  : out std_logic);
  end component;
```

```
begin
   GBUF_FOR_MUX_CLOCK : BUFGMUX
     port map(
       I0 => SLOW_CLOCK,
       I1 => FAST_CLOCK,
       S  => SEL,
       O  => CLOCK_GBUF);
   Data_Path : process (CLOCK_GBUF)
     begin
       if (CLOCK_GBUF'event and CLOCK_GBUF='1')then
           DOUT <= DATA;
       end if;
     end process;
end XILINX;
```

- Verilog example

```
///////////////////////////////////////////////
// CLOCK_MUX_BUFG.V Version 1.2              //
// This is an example of an instantiation of //
// a multiplexing global buffer (BUFGMUX)    //
// from an internally driven signal          //
///////////////////////////////////////////////

module clock_mux
  (DATA,SEL,SLOW_CLOCK,FAST_CLOCK,DOUT);

  input DATA, SEL, SLOW_CLOCK, FAST_CLOCK;
  output DOUT;

  reg CLOCK, DOUT;
  wire CLOCK_GBUF;


  BUFGMUX GBUF_FOR_MUX_CLOCK
    (.O(CLOCK_GBUF),
     .I0(SLOW_CLOCK),
     .I1(FAST_CLOCK),
     .S(SEL));

  always @ (posedge CLOCK_GBUF)
      DOUT <= DATA;

endmodule
```

# Using Advanced Clock Management

Virtex™/E, and Spartan-II™ devices feature Clock Delay-Locked Loop (CLKDLL) for advanced clock management. The CLKDLL can eliminate skew between the clock input pad and internal clock-input pins throughout the device. CLKDLL also provides four quadrature phases of the source clock. With CLKDLL you can eliminate clock-distribution delay, double the clock, or divide the clock. The CLKDLL also operates as a clock mirror. By driving the output from a DLL off-chip and then back on again, the CLKDLL can be used to de-skew a board level clock among multiple Virtex™, Virtex-E™ and Spartan-II™ devices. For detailed information on using CLKDLLs, refer to the *Libraries Guide* and the XAPP132 and XAPP174 Application Notes.

In Virtex™-II / II Pro / II Pro X or Spartan-3™ devices, the Digital Clock Manager (DCM) is available for advanced clock management. The DCM contains the following four main features. For more information on the functionality of these features, refer to the *Libraries Guide* and the *Virtex-II Platform FPGA User Guide*.

- *Delay Locked Loop (DLL)* — The DLL feature is very similar to CLKDLL.
- *Digital Phase Shifter (DPS)* — The DPS provides a clock shifted by a fixed or variable phase skew.
- *Digital Frequency Synthesizer (DFS)* — The DFS produces a wide range of possible clock frequencies related to the input clock.

*Table 4-2:* **CLKDLL and DCM Resources**

| | **Virtex™/ Spartan-II™** | **Virtex-E™** | **Virtex™-II/II Pro/ II Pro X/ Spartan-3™** |
|---|---|---|---|
| CLKDLL | 4 | 8 | N / A |
| DCM | N / A | N / A | 4 - 12 |

## Using CLKDLL (Virtex™, Virtex-E™ and Spartan-II™)

There are four CLKDLLs in each Virtex™ / Spartan-II™ device and eight in each Virtex-E™ device. There are also four global clock input buffers (IBUFG) in the Virtex™ / E and Spartan-II™ devices to bring external clocks in to the CLKDLL. The following VHDL / Verilog example shows a possible connection and usage of CLKDLL in your design. Cascading three CLKDLLs in the Virtex™ / Spartan-II™ device is not allowed due to excessive jitter.

Synthesis tools do not infer CLKDLLs. The following examples show how to instantiate CLKDLLs in your VHDL and Verilog code.

- VHDL example

```
library IEEE;
use IEEE.std_logic_1164.all;
entity CLOCK_TEST is
  port(
    ACLK         : in std_logic;
-- off chip feedback, connected to OUTBCLK on the board.
    BCLK         : in std_logic;
--OUT CLOCK
    OUTBCLK      : out std_logic;
    DIN          : in std_logic_vector(1 downto 0);
    RESET        : in std_logic;
    QOUT         : out std_logic_vector (1 downto 0);
-- CLKDLL lock signal
    BCLK_LOCK    : out std_logic
      );
end CLOCK_TEST;
architecture RTL of CLOCK_TEST is
  component IBUFG
    port (
      I : in std_logic;
      O : out std_logic);
  end component;
  component BUFG
    port (
      I : in std_logic;
      O : out std_logic);
  end component;
  component CLKDLL
    port (
        CLKIN  : in std_logic;
        CLKFB  : in std_logic;
        RST    : in std_logic;
        CLK0   : out std_logic;
        CLK90  : out std_logic;
        CLK180 : out std_logic;
        CLK270 : out std_logic;
        CLKDV  : out std_logic;
        CLK2X  : out std_logic;
    LOCKED : out std_logic);
  end component;
  -- Clock signals
  signal ACLK_ibufg     : std_logic;
  signal BCLK_ibufg     : std_logic;
  signal ACLK_2x        : std_logic;
  signal ACLK_2x_design : std_logic;
  signal ACLK_lock      : std_logic;
begin
  ACLK_ibufg_inst : IBUFG
    port map (
      I => ACLK,
      O => ACLK_ibufg
        );
  BCLK_ibufg_inst : IBUFG
    port map (
      I => BCLK,
      O => BCLK_ibufg
        );
  ACLK_bufg : BUFG
    port map (
      I => ACLK_2x,
      O => ACLK_2x_design
        );
  ACLK_dll : CLKDLL
    port map (
        CLKIN  => ACLK_ibufg,
        CLKFB  => ACLK_2x_design,
        RST    => ’0’,
        CLK2X  => ACLK_2x,
        CLK0   => OPEN,
        CLK90  => OPEN,
        CLK180 => OPEN,
        CLK270 => OPEN,
        CLKDV  => OPEN,
        LOCKED => ACLK_lock
          );
```

```
BCLK_dll_out : CLKDLL
   port map (
      CLKIN  => ACLK_ibufg,
      CLKFB  => BCLK_ibufg,
      RST    => ’0’,
      CLK2X  => OUTBCLK,
      CLK0   => OPEN,
      CLK90  => OPEN,
      CLK180 => OPEN,
      CLK270 => OPEN,
      CLKDV  => OPEN,
      LOCKED => BCLK_lock
        );
process (ACLK_2x_design, RESET)
begin
  if RESET = ’1’ then
     QOUT <= "00";
  elsif ACLK_2x_design’event and ACLK_2x_design = ’1’ then
     if ACLK_lock = ’1’ then
        QOUT <= DIN;
     end if;
  end if;
end process;
END RTL;
```

- Verilog example

```
// Verilog Example
// In this example ACLK’s frequency is doubled,
// used inside and outside the chip.
// BCLK and OUTBCLK are connected in the board
// outside the chip.

module clock_test(ACLK, DIN, QOUT, BCLK, OUTBCLK, BCLK_LOCK, RESET);
  input  ACLK, BCLK;
  input  RESET;
  input  [1:0] DIN;
  output [1:0] QOUT;
  output OUTBCLK, BCLK_LOCK;
  reg [1:0] QOUT;

IBUFG CLK_ibufg_A
   (.I (ACLK),
    .O(ACLK_ibufg)
     );
BUFG ACLK_bufg
   (.I (ACLK_2x),
    .O (ACLK_2x_design)
     );
IBUFG CLK_ibufg_B
   (.I (BCLK),     // connected to OUTBCLK outside
    .O(BCLK_ibufg)
     );
```

```
CLKDLL ACLK_dll_2x   // 2x clock
    (.CLKIN(ACLK_ibufg),
     .CLKFB(ACLK_2x_design),
     .RST(1’b0),
     .CLK2X(ACLK_2x),
     .CLK0(),
     .CLK90(),
     .CLK180(),
     .CLK270(),
     .CLKDV(),
     .LOCKED(ACLK_lock)
      );
CLKDLL BCLK_dll_OUT // off-chip synchronization
    (.CLKIN(ACLK_ibufg),
     .CLKFB(BCLK_ibufg), // BCLK and OUTBCLK is
                        // connected outside the
                        // chip.
     .RST(1’b0),
     .CLK2X(OUTBCLK),   //connected to BCLK outside
     .CLK0(),
     .CLK90(),
     .CLK180(),
     .CLK270(),
     .CLKDV(),
     .LOCKED(BCLK_LOCK)
      );

always @(posedge ACLK_2x_design or posedge RESET)
  begin
    if (RESET)
        QOUT[1:0] <= 2’b00;
    else if (ACLK_lock)
        QOUT[1:0] <= DIN[1:0];
  end
endmodule
```

## Using the Additional CLKDLL in Virtex-E™

There are eight CLKDLLs in each Virtex-E™ device, with four located at the top and four at the bottom as shown in Figure 4-1. The basic operations of the DLLs in the Virtex-E™ devices remain the same as in the Virtex™ and Spartan-II™ devices, but the connections may have changed for some configurations.

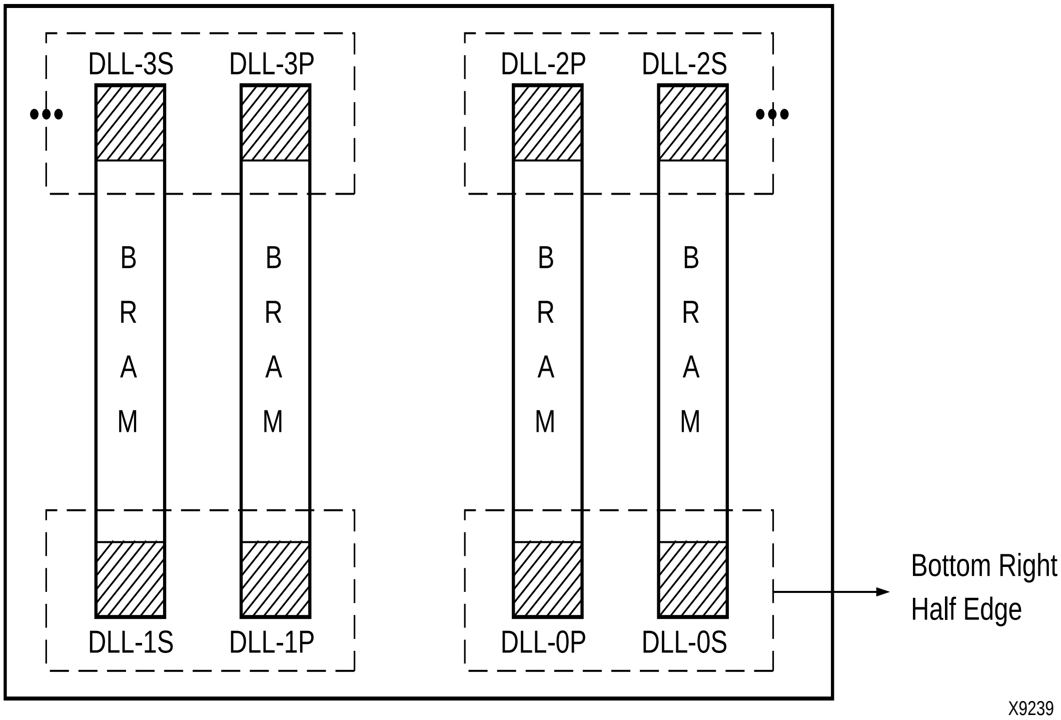


*Figure 4-1:* **DLLs in Virtex-E™ Devices**

Two DLLs located in the same half-edge (top-left, top-right, bottom-right, bottom-left) can be connected together, without using a BUFG between the CLKDLLs, to generate a 4x clock as shown in Figure 4-2.

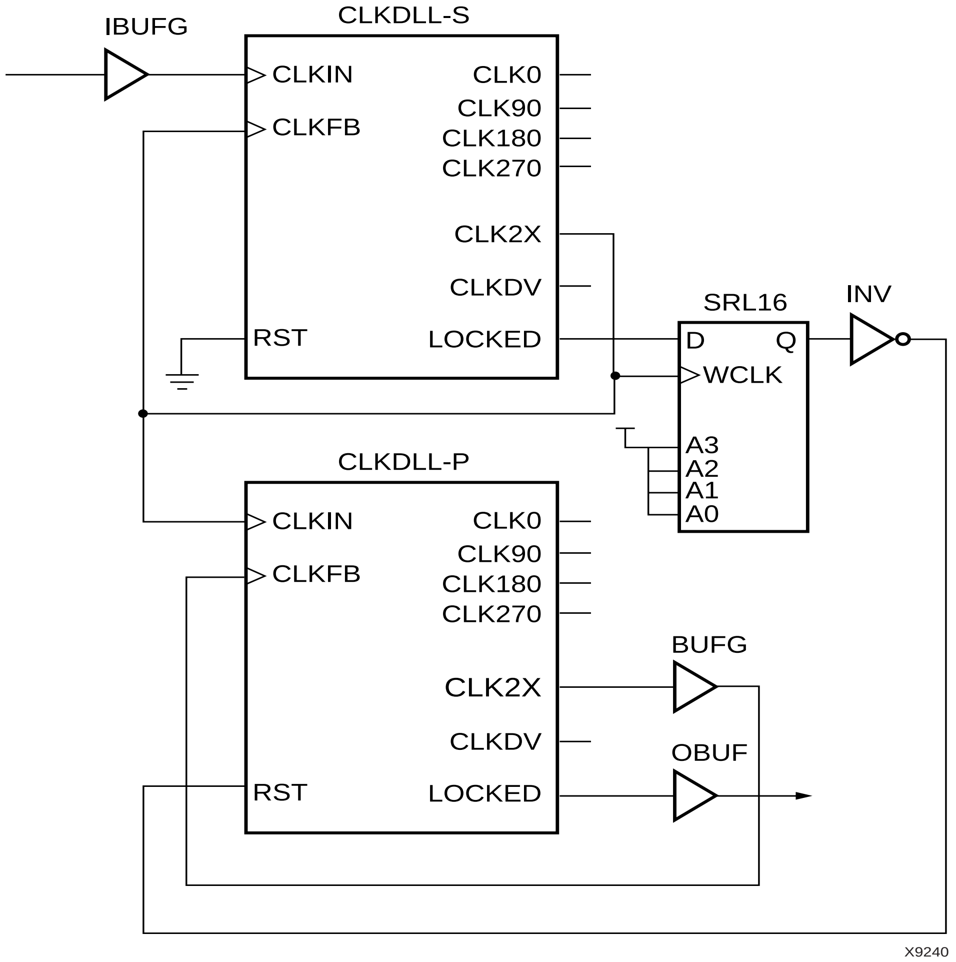


*Figure 4-2:* **DLL Generation of 4x Clock in Virtex-E™ Devices**

Following are examples of coding a CLKDLL in both VHDL and Verilog.

- VHDL example

```
library IEEE;
use IEEE.std_logic_1164.all;
entity CLOCK_TEST is
  port(
        ACLK  : in std_logic;
        DIN   : in std_logic_vector(1 downto 0);
        RESET : in std_logic;
        QOUT  : out std_logic_vector (1 downto 0);
    -- CLKDLL lock signal
        BCLK_LOCK : out std_logic
        );
end CLOCK_TEST;
```

```
architecture RTL of CLOCK_TEST is
  component IBUFG
    port (
        I : in std_logic;
        O : out std_logic);
  end component;
  component BUFG
    port (
        I : in std_logic;
        O : out std_logic);
  end component;
  component CLKDLL
    port (
        CLKIN  : in std_logic;
        CLKFB  : in std_logic;
        RST    : in std_logic;
        CLK0   : out std_logic;
        CLK90  : out std_logic;
        CLK180 : out std_logic;
        CLK270 : out std_logic;
        CLKDV  : out std_logic;
        CLK2X  : out std_logic;
        LOCKED : out std_logic);
  end component;
  -- Clock signals
  signal ACLK_ibufg        : std_logic;
  signal ACLK_2x, BCLK_4x  : std_logic;
  signal BCLK_4x_design    : std_logic;
  signal BCLK_lockin       : std_logic;
begin
  ACLK_ibufginst : IBUFG
    port map (
        I => ACLK,
        O => ACLK_ibufg
        );
  BCLK_bufg: BUFG
    port map (
        I => BCLK_4x,
        O => BCLK_4x_design);
  ACLK_dll : CLKDLL
    port map (
        CLKIN  => ACLK_ibufg,
        CLKFB  => ACLK_2x,
        RST    => ’0’,
        CLK2X  => ACLK_2x,
        CLK0   => OPEN,
        CLK90  => OPEN,
        CLK180 => OPEN,
        CLK270 => OPEN,
        CLKD   => OPEN,
        LOCKED => OPEN
          );
  BCLK_dll : CLKDLL
    port map (
        CLKIN   => ACLK_2x,
        CLKFB   => BCLK_4x_design,
        RST     => ’0’,
        CLK2X   => BCLK_4x,
        CLK0    => OPEN,
        CLK90   => OPEN,
        CLK180  => OPEN,
        CLK270  => OPEN,
        CLKDV   => OPEN,
        LOCKED  => BCLK_lockin
          );
process (BCLK_4x_design, RESET)
begin
  if RESET = ’1’ then
     QOUT <= "00";
  elsif BCLK_4x_design’event and BCLK_4x_design = ’1’ then
     if BCLK_lockin = ’1’ then
          QOUT <= DIN;
     end if;
  end if;
end process;
  BCLK_lock <= BCLK_lockin;
END RTL;
```

- Verilog example

```
module clock_test(ACLK, DIN, QOUT, BCLK_LOCK, RESET);
  input  ACLK;
  input  RESET;
  input  [1:0] DIN;
  output [1:0] QOUT;
  output BCLK_LOCK;
  reg    [1:0] QOUT;
IBUFG CLK_ibufg_A
       (.I (ACLK),
        .O(ACLK_ibufg)
        );
BUFG BCLK_bufg
       (.I (BCLK_4x),
        .O (BCLK_4x_design)
        );
CLKDLL ACLK_dll_2x   // 2x clock
       (.CLKIN(ACLK_ibufg),
        .CLKFB(ACLK_2x),
        .RST(1’b0),
        .CLK2X(ACLK_2x),
        .CLK0(),
        .CLK90(),
        .CLK180(),
        .CLK270(),
        .CLKDV(),
        .LOCKED()
        );
```

```
CLKDLL BCLK_dll_4x    // 4x clock
       (.CLKIN(ACLK_2x),
        .CLKFB(BCLK_4x_design),   // BCLK_4x after bufg
        .RST(1'b0),
        .CLK2X(BCLK_4x),
        .CLK0(),
        .CLK90(),
        .CLK180(),
        .CLK270(),
        .CLKDV(),
        .LOCKED(BCLK_LOCK)
         );
always @(posedge BCLK_4x_design or posedge RESET)
  begin
    if (RESET)
          QOUT[1:0] <= 2'b00;
    else if (BCLK_LOCK)
          QOUT[1:0] <= DIN[1:0];
  end
endmodule
```

## Using BUFGDLL

BUFGDLL macro is the simplest way to ensure zero propagation delay for a high-fanout on-chip clock from the external input. This macro uses the IBUFG, CLKDLL and BUFG primitive to implement the most basic DLL application as shown in Figure 4-3.

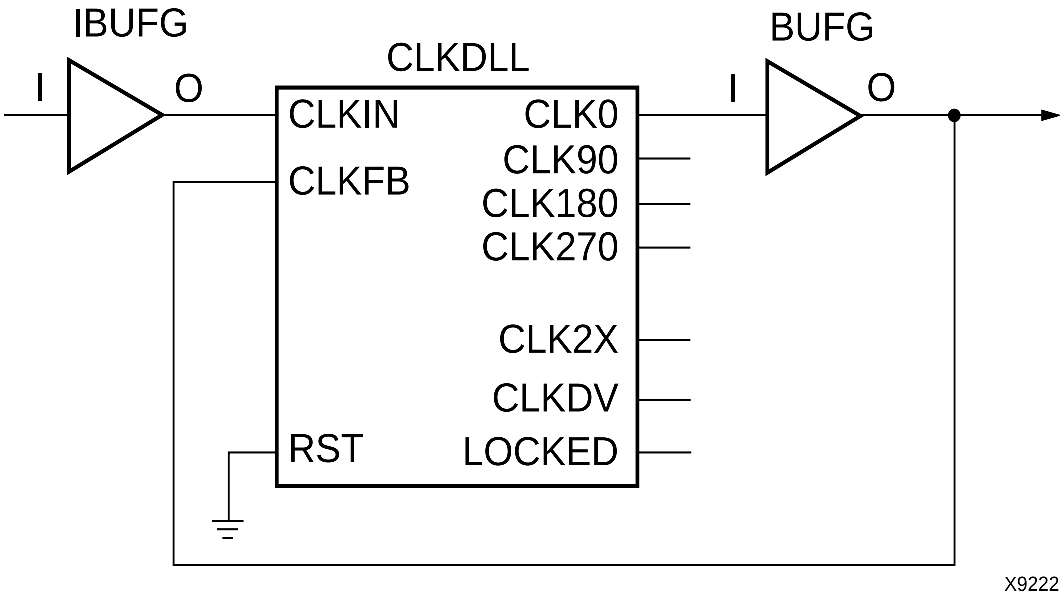


*Figure 4-3:* **BUFGDLL Schematic**

In FPGA Compiler II™, use the Constraints Editor to change the global buffer insertion to BUFGDLL.

In LeonardoSpectrum™, set the following attribute in the command line or TCL script.

**set_attribute -port** *CLOCK_PORT* **-name PAD -value BUFGDLL**

LeonardoSpectrum™ supports implementation of BUFGDLL with the CLKDLLHF component. To use this implementation, set the following attribute.

**set_attribute -port** *CLOCK_PORT* **-name PAD -value BUFGDLLHF**

In Synplify™, set the following attribute in the SDC file.

**define_attribute** *port_name* **xc_clockbuftype {BUFGDLL}**

This attribute can be applied to the clock port in HDL code as well.

In XST, the BUFGDLL can be used by the CLOCK_BUFFER constraint entered in either HDL or the XST constraints file. For more information on using XST specific constraints see the *XST User Guide*.

## CLKDLL Attributes

To specify how the signal on the CLKDIV pin is frequency divided with respect to the CLK0 pin, you can set the CLKDV_DIVIDE property. The values allowed for this property are 1.5, 2, 2.5, 3, 4, 5, 8, or 16. The default is 2.

In HDL code, the CLKDV_DIVIDE property is set as an attribute to the CLKDLL instance.

The following are VHDL and Verilog coding examples of CLKDLL attributes.

- VHDL example

```
library IEEE;
use IEEE.std_logic_1164.all;
entity CLOCK_TEST is
   port(
       ACLK  : in  std_logic;
       DIN   : in  std_logic_vector(1 downto 0);
       RESET : in  std_logic;
       QOUT  : out std_logic_vector (1 downto 0)
        );
end CLOCK_TEST;
architecture RTL of CLOCK_TEST is
  component IBUFG
   port (
       I : in  std_logic;
       O : out std_logic);
  end component;
  component BUFG
   port (
       I : in  std_logic;
       O : out std_logic);
  end component;
  component CLKDLL
   port (
       CLKIN  : in std_logic;
       CLKFB  : in std_logic;
       RST    : in std_logic;
       CLK0   : out std_logic;
       CLK90  : out std_logic;
       CLK180 : out std_logic;
       CLK270 : out std_logic;
       CLKDV  : out std_logic;
       CLK2X  : out std_logic;
       LOCKED : out std_logic
        );
end component;
-- Clock signals
signal ACLK_ibufg            : std_logic;
signal div_2, div_2_design   : std_logic;
signal ACLK0, ACLK0bufg      : std_logic;
signal logic_0               : std_logic;

attribute CLKDV_DIVIDE: string;
attribute CLKDV_DIVIDE of ACLK_dll : label is "2";

logic_0 <= ‘0’;

begin
  ACLK_ibufginst : IBUFG
   port map (
       I => ACLK,
       O => ACLK_ibufg
        );
  ACLK_bufg: BUFG
   port map (
       I => ACLK0,
       O => ACLK0bufg
       );
  DIV_bufg: BUFG
   port map (
       I => div_2,
       O => div_2_design
       );
  ACLK_dll : CLKDLL
   port map (
       CLKIN  => ACLK_ibufg,
       CLKFB  => ACLK0bufg,
       RST    => logic_0,
       CLK2X  => OPEN,
       CLK0   => ACLK0,
       CLK90  => OPEN,
       CLK180 => OPEN,
       CLK270 => OPEN,
       CLKDV  => div_2,
       LOCKED => OPEN
       );
```

```
process (div_2_design, RESET)
begin
  if RESET = ’1’ then
      QOUT <= "00";
  elsif div_2_design’event and div_2_design = ’1’ then
      QOUT <= DIN;
  end if;
end process;
END RTL;
```

- Verilog example

```
module clock_test(ACLK, DIN, QOUT, RESET);
  input   ACLK;
  input   RESET;
  input   [1:0] DIN;
  output  [1:0] QOUT;
  reg     [1:0] QOUT;
  IBUFG CLK_ibufg_A
        (.I (ACLK),
         .O(ACLK_ibufg)
          );
  BUFG div_CLK_bufg
        (.I (div_2),
         .O (div_2_design)
          );
  BUFG clk0_bufg ( .I(clk0), .O(clk_bufg));
  CLKDLL ACLK_div_2          // div by 2
       (.CLKIN(ACLK_ibufg),
        .CLKFB(clk_bufg),
        .RST(1’b0),
        .CLK2X(),
        .CLK0(clk0),
        .CLK90(),
        .CLK180(),
        .CLK270(),
        .CLKDV(div_2),
        .LOCKED()
        );
//exemplar attribute ACLK_div_2 CLKDV_DIVIDE 2
//synopsys attribute CLKDV_DIVIDE “2”
//synthesis attribute CLKDV_DIVIDE of ACLK_div_2 is “2”
always @(posedge div_2_design or posedge RESET)
begin
  if (RESET)
    QOUT <= 2’b00;
  else
    QOUT <= DIN;
  end
endmodule
```

## Using DCM In Virtex™-II/II Pro/II Pro X and Spartan-3™

Use the DCM in your Virtex-II™, Virtex-II Pro™, Virtex-II Pro X™ or Spartan-3™ design to improve routing between clock pads and global buffers. Most synthesis tools currently do not automatically infer the DCM. Hence, you need to instantiate the DCM in your VHDL and Verilog designs.

To more easily set up the DCM, use the Clocking Wizard. See "Architecture Wizard" in Chapter 2 for details on the Clocking Wizard.

Please refer to the Design Considerations Chapter of the *Virtex-II Platform FPGA User Guide* or the *Virtex-II Pro Platform FPGA User Guide*, respectively, for information on the various features in the DCM. These books can be found on the Xilinx® website at http://www.xilinx.com.

The following examples show how to instantiate DCM and apply a DCM attribute in VHDL and Verilog.

***Note:*** For more information on passing attributes in the HDL code to different synthesis vendors, refer to Chapter 3, "General HDL Coding Styles".

- VHDL example

```
-- Using a DCM for Virtex-II (VHDL)
--
-- The following code passes the attribute for
-- the synthesis tools Synplify, FPGA Compiler II
-- LeonardoSpectrum and XST.
library IEEE;
use IEEE.std_logic_1164.all;
entity clock_block is
  port (
    CLK_PAD             : in std_logic;
    SPREAD_SPECTRUM_YES : in std_logic;
    RST_DLL             : in std_logic;
    CLK_out             : out std_logic;
    LOCKED              : out std_logic
    );
end clock_block;
architecture STRUCT of clock_block is
  signal CLK, CLK_int, CLK_dcm : std_logic;
  attribute CLKIN_PERIOD : string;
  attribute CLKIN_PERIOD of U2 : label is "10";
  component IBUFG
    port (
        I : in std_logic;
        O : out std_logic);
  end component;
  component BUFG
    port (
        I : in std_logic;
        O : out std_logic);
  end component;
  component DCM is
    port (
        CLKFB    : in std_logic;
        CLKIN    : in std_logic;
        DSSEN    : in std_logic;
        PSCLK    : in std_logic;
        PSEN     : in std_logic;
        PSINCDEC : in std_logic;
        RST      : in std_logic;
        CLK0     : out std_logic;
        CLK90    : out std_logic;
        CLK180   : out std_logic;
        CLK270   : out std_logic;
        CLK2X    : out std_logic;
        CLK2X180 : out std_logic;
        CLKDV    : out std_logic;
        CLKFX    : out std_logic;
        CLKFX180 : out std_logic;
        LOCKED   : out std_logic;
        PSDONE   : out std_logic;
        STATUS   : out std_logic_vector (7 downto 0)
        );
  end component;
signal logic_0 : std_logic;

begin
  logic_0 <= ‘0’;

  U1 : IBUFG port map ( I => CLK_PAD, O => CLK_int);
  U2 : DCM port map (
        CLKFB    => CLK,
        CLKIN    => CLK_int,
        DSSEN    => logic_0,
        PSCLK    => logic_0,
        PSEN     => logic_0,
        PSINCDEC => logic_0,
        RST      => RST_DLL,
        CLK0     => CLK_dcm,
        LOCKED   => LOCKED
        );
  U3 : BUFG port map (I => CLK_dcm, O => CLK);
  CLK_out <= CLK;
end
end architecture STRUCT;
```

- Verilog example

```
// Using a DCM for Virtex-II (Verilog)
//
// The following code passes the attribute for the
// synthesis tools Synplify, FPGA Compiler II,
// LeonardoSpectrum and XST.
module clock_top (clk_pad,rst_dll, clk_out,locked);
  input  clk_pad, spread_spectrum_yes, rst_dll;
  output clk_out, locked;
  wire   clk, clk_int, clk_dcm;
  IBUFG u1 (.I (clk_pad), .O (clk_int));
  DCM u2(.CLKFB  (clk),
       .CLKIN    (clk_int),
       .DSSEN    (spread_spectrum_yes),
       .PSCLK    (1'b0),
       .PSEN     (1'b0),
       .PSINCDEC (1'b0),
       .RST      (rst_dll),
       .CLK0     (clk_dcm),
       .LOCKED    (locked))
/* synthesis CLKIN_PERIOD = "10" */;
// synopsys attribute CLKIN_PERIOD 10
// exemplar attribute u2 CLKIN_PERIOD 10
// synthesis attribute CLKIN_PERIOD of u2 is "10"
  BUFG u3(.I (clk_dcm), .O (clk));
  assign clk_out = clk;
endmodule // clock_top
```

## Attaching Multiple Attributes to CLKDLL and DCM

CLKDLLs and DCMs can be configured to various modes by attaching attributes during instantiation. In some cases, multiple attributes must be attached to get the desired configuration. The following HDL coding examples show how to attach multiple attributes to DCM components. The same method can be used to attach attributes to CLKDLL components.

See the *Libraries Guide* for available attributes for Virtex™/Virtex-E™ CLKDLL. See the *Virtex-II Platform FPGA User Guide* for the available attributes for Virtex™-II/II Pro/ II Pro X and Spartan-3™ DCM.

- VHDL example for Synplify™

  This example attaches multiple attributes to DCM components using the Synplify™ XC_PROP attribute.

  ***Note:*** Do not insert carriage returns between the values assigned to xc_props. A carriage return could cause Synplify™ to attach only part of the attributes.

```
-- VHDL code begin --
library IEEE;
library virtex2;
use IEEE.std_logic_1164.all;
use virtex2.components.all;
```

```
entity DCM_TOP is
  port(
       clock_in          : in std_logic;
       clock_out         : out std_logic;
       clock_with_ps_out : out std_logic;
       reset             : out std_logic
       );
end DCM_TOP;

architecture XILINX of DCM_TOP is
  signal low, high      : std_logic;
  signal dcm0_locked    : std_logic;
  signal dcm1_locked    : std_logic;
  signal clock          : std_logic;
  signal clk0           : std_logic;
  signal clk1           : std_logic;
  signal clock_with_ps : std_logic;
  signal clock_out_int : std_logic;

  attribute xc_props : string;
  attribute xc_props of dcm0: label is "DLL_FREQUENCY_MODE =
LOW,DUTY_CYCLE_CORRECTION = TRUE,STARTUP_WAIT =
TRUE,DFS_FREQUENCY_MODE = LOW,CLKFX_DIVIDE = 1,CLKFX_MULTIPLY =
1,CLK_FEEDBACK = 1X,CLKOUT_PHASE_SHIFT = NONE,PHASE_SHIFT = 0";
-- Do not insert any carriage return between the
-- lines above.
attribute xc_props of dcm1: label is "DLL_FREQUENCY_MODE
=LOW,DUTY_CYCLE_CORRECTION = TRUE,STARTUP_WAIT =
TRUE,DFS_FREQUENCY_MODE = LOW,CLKFX_DIVIDE = 1,CLKFX_MULTIPLY =
1,CLK_FEEDBACK = 1X,CLKOUT_PHASE_SHIFT = FIXED,PHASE_SHIFT = 0";
-- Do not insert any carriage return between the lines above.
begin
  low <= ’0’;
  high <= ’1’;
  reset <= not(dcm0_locked and dcm1_locked);
  clock_with_ps_out <= clock_with_ps;
  clock_out <= clock_out_int;

  U1 : IBUFG port map ( I => clock_in, O => clock);

  dcm0 : DCM port map (
       CLKFB => clock_out_int,
       CLKIN => clock,
       DSSEN => low,
       PSCLK => low,
       PSEN => low,
       PSINCDEC => low,
       RST => low,
       CLK0 => clk0,
       LOCKED => dcm0_locked
       );
  clk_buf0 : BUFG port map (I => clk0, O => clock_out_int);
  dcm1: DCM port map (
       CLKFB    => clock_with_ps,
       CLKIN    => clock,
       DSSEN    => low,
       PSCLK    => low,
       PSEN     => low,
       PSINCDEC => low,
       RST      => low,
       CLK0     => clk1,
       LOCKED   => dcm1_locked
       );
  clk_buf1 : BUFG port map(
       I => clk1,
       O => clock_with_ps
       );
end XILINX;
```

- Verilog example for Synplify™

  This example attaches multiple attributes to DCM components using the Synplify™ XC_PROP attribute.

  ***Note:*** Do not insert carriage returns between the values assigned to xc_props. A carriage return could cause Synplify™ to attach only part of the attributes.

```
//Verilog code begin
‘include “/path_to/virtex2.v”
module DCM_TOP(
       clock_in,
       clock_out,
       clock_with_ps_out,
       reset
       );

input clock_in;
output clock_out;
output clock_with_ps_out;
output reset;

wire low;
wire high;
wire dcm0_locked;
wire dcm1_locked;
wire reset;
wire clk0;
wire clk1;

assign low = 1'b0;
assign high = 1'b1;
assign reset = (dcm0_locked ~& dcm1_locked);
IBUFG CLOCK_IN (
       .I(clock_in),
       .O(clock)
       );
```

```
DCM DCM0 (
       .CLKFB(clock_out),
       .CLKIN(clock),
       .DSSEN(low),
       .PSCLK(low),
       .PSEN(low),
       .PSINCDEC(low),
       .RST(low),
       .CLK0(clk0),
       .CLK90(),
       .CLK180(),
       .CLK270(),
       .CLK2X(),
       .CLK2X180(),
       .CLKDV(),
       .CLKFX(),
       .CLKFX180(),
       .LOCKED(dcm0_locked),
       .PSDONE(),
       .STATUS()
       )
/*synthesis xc_props="DLL_FREQUENCY_MODE = LOW,DUTY_CYCLE_CORRECTION =
TRUE,STARTUP_WAIT = TRUE,DFS_FREQUENCY_MODE = LOW,CLKFX_DIVIDE =
1,CLKFX_MULTIPLY = 1,CLK_FEEDBACK = 1X,CLKOUT_PHASE_SHIFT =
NONE,PHASE_SHIFT = 0" */;
//Do not insert any carriage return between the
//lines above.

BUFG CLK_BUF0(
       .O(clock_out),
       .I(clk0),
       );

DCM DCM1 (
       .CLKFB(clock_with_ps_out),
       .CLKIN(clock),
       .DSSEN(low),
       .PSCLK(low),
       .PSEN(low),
       .PSINCDEC(low),
       .RST(low),
       .CLK0(clk1),
       .CLK90(),
       .CLK180(),
       .CLK270(),
       .CLK2X(),
       .CLK2X180(),
       .CLKDV(),
       .CLKFX(),
       .CLKFX180(),
       .LOCKED(dcm1_locked),
       .PSDONE(),
       .STATUS()
       )
```

```
/*synthesis xc_props="DLL_FREQUENCY_MODE =LOW,DUTY_CYCLE_CORRECTION =
TRUE,STARTUP_WAIT = TRUE,DFS_FREQUENCY_MODE = LOW,CLKFX_DIVIDE =
1,CLKFX_MULTIPLY = 1,CLK_FEEDBACK = 1X,CLKOUT_PHASE_SHIFT =
FIXED,PHASE_SHIFT = 0"   */;
//Do not insert any carriage return between the
//lines above.

BUFG CLK_BUF1(
    .O(clock_with_ps_out),
    .I(clk1)
    );

//The following Verilog code is for simulation only
//synthesis translate_off
defparam DCM0.DLL_FREQUENCY_MODE = "LOW";
defparam DCM0.DUTY_CYCLE_CORRECTION = "TRUE";
defparam DCM0.STARTUP_WAIT = "TRUE";
defparam DCM0.DFS_FREQUENCY_MODE = "LOW";
defparam DCM0.CLKFX_DIVIDE = 1;
defparam DCM0.CLKFX_MULTIPLY = 1;
defparam DCM0.CLK_FEEDBACK = "1X";
defparam DCM0.CLKOUT_PHASE_SHIFT = "NONE";
defparam DCM0.PHASE_SHIFT = "0";

defparam DCM1.DLL_FREQUENCY_MODE = "LOW";
defparam DCM1.DUTY_CYCLE_CORRECTION = "TRUE";
defparam DCM1.STARTUP_WAIT = "TRUE";
defparam DCM1.DFS_FREQUENCY_MODE = "LOW";
defparam DCM1.CLKFX_DIVIDE = 1;
defparam DCM1.CLKFX_MULTIPLY = 1;
defparam DCM1.CLK_FEEDBACK = "1X";
defparam DCM1.CLKOUT_PHASE_SHIFT = "FIXED";
defparam DCM1.PHASE_SHIFT = "0";
//synthesis translate_on
endmodule // DCM_TOP
```

- VHDL example for LeonardoSpectrum™

```
library IEEE;
use IEEE.std_logic_1164.all;

entity DCM_TOP is
    port (
        clock_in : in std_logic;
        clock_out : out std_logic;
        clock_with_ps_out : out std_logic;
        reset : out std_logic
        );
end DCM_TOP;

architecture XILINX of DCM_TOP is
signal low, high : std_logic;
signal dcm0_locked : std_logic;
signal dcm1_locked : std_logic;
signal clock : std_logic;
signal clk0 : std_logic;
signal clk1 : std_logic;
signal clock_with_ps : std_logic;
signal clock_out_int : std_logic;
```

```
attribute DLL_FREQUENCY_MODE : string;
attribute DUTY_CYCLE_CORRECTION : string;
attribute STARTUP_WAIT : string;
attribute DFS_FREQUENCY_MODE : string;
attribute CLKFX_DIVIDE : string;
attribute CLKFX_MULTIPLY : string;
attribute CLK_FEEDBACK : string;
attribute CLKOUT_PHASE_SHIFT : string;
attribute PHASE_SHIFT : string;

attribute DLL_FREQUENCY_MODE of dcm0 : label is "LOW";
attribute DUTY_CYCLE_CORRECTION of dcm0 : label is "TRUE";
attribute STARTUP_WAIT of dcm0 : label is "TRUE";
attribute DFS_FREQUENCY_MODE of dcm0: label is "LOW";
attribute CLKFX_DIVIDE of dcm0 : label is "1";
attribute CLKFX_MULTIPLY of dcm0 : label is "1";
attribute CLK_FEEDBACK of dcm0 : label is "1X";
attribute CLKOUT_PHASE_SHIFT of dcm0 : label is "NONE";
attribute PHASE_SHIFT of dcm0 : label is "0";

attribute DLL_FREQUENCY_MODE of dcm1 : label is "LOW";
attribute DUTY_CYCLE_CORRECTION of dcm1 : label is "TRUE";
attribute STARTUP_WAIT of dcm1 : label is "TRUE";
attribute DFS_FREQUENCY_MODE of dcm1 : label is "LOW";
attribute CLKFX_DIVIDE of dcm1 : label is "1";
attribute CLKFX_MULTIPLY of dcm1 : label is "1";
attribute CLK_FEEDBACK of dcm1 : label is "1X";
attribute CLKOUT_PHASE_SHIFT of dcm1 : label is "FIXED";
attribute PHASE_SHIFT of dcm1 : label is "0";

component IBUFG is
   port (
      I : in std_logic;
      O : out std_logic
      );
end component;

component BUFG is
   port (
      I : in std_logic;
      O : out std_logic
      );
end component;
```

```
component DCM is
    port (
        CLKFB : in std_logic;
        CLKIN : in std_logic;
        DSSEN : in std_logic;
        PSEN : in std_logic;
        PSINCDEC : in std_logic;
        RST : in std_logic;
        CLK0 : out std_logic;
        CLK90 : out std_logic;
        CLK180 : out std_logic;
        CLK270 : out std_logic;
        CLK2X : out std_logic;
        CLK2X180 : out std_logic;
        CLKDV : out std_logic;
        CLKFX : out std_logic;
        CLKFX180 : out std_logic;
        LOCKED : out std_logic;
        PSDONE : out std_logic;
        STATUS : out std_logic_vector (7 downto 0)
        );
end component;

begin
  low <= ’0’;
  high <= ’1’;
  reset <= not(dcm0_locked and dcm1_locked);
  clock_with_ps_out <= clock_with_ps;
  clock_out <= clock_out_int;

  U1 : IBUFG port map ( I => clock_in, O => clock);

  dcm0 : DCM port map (
        CLKFB => clock_out_int,
        CLKIN => clock,
        DSSEN => low,
        PSCLK => low,
        PSEN => low,
        PSINCDEC => low,
        RST => low,
        CLK0 => clk0,
        LOCKED => dcm0_locked
        );
  clk_buf0 : BUFG port map (I => clk0, O => clock_out_int);

  dcm1: DCM port map (
        CLKFB    => clock_with_ps,
        CLKIN    => clock,
        DSSEN    => low,
        PSCLK    => low,
        PSEN     => low,
        PSINCDEC => low,
        RST      => low,
        CLK0     => clk1,
        LOCKED   => dcm1_locked
        );

  clk_buf1: BUFG port map(
        I => clk1,
        O => clock_with_ps
        );
end XILINX;
```

- Verilog example for LeonardoSpectrum™

```
module DCM_TOP(
       clock_in,
       clock_out,
       clock_with_ps_out,
       reset
       );

input clock_in;
output clock_out;
output clock_with_ps_out;
output reset;

wire low;
wire high;
wire dcm0_locked;
wire dcm1_locked;
wire reset;
wire clk0;
wire clk1;

assign low = 1’b0;
assign high = 1’b1;
assign reset = (dcm0_locked ~& dcm1_locked);

IBUFG CLOCK_IN (
       .I(clock_in),
       .O(clock)
       );
```

```
DCM DCM0 (
       .CLKFB(clock_out),
       .CLKIN(clock),
       .DSSEN(low),
       .PSCLK(low),
       .PSEN(low),
       .PSINCDEC(low),
       .RST(low),
       .CLK0(clk0),
       .CLK90(),
       .CLK180(),
       .CLK270(),
       .CLK2X(),
       .CLK2X180(),
       .CLKDV(),
       .CLKFX(),
       .CLKFX180(),
       .LOCKED(dcm0_locked),
       .PSDONE(),
       .STATUS()
       );
//exemplar attribute DCM0 DLL_FREQUENCY_MODE LOW
//exemplar attribute DCM0 DUTY_CYCLE_CORRECTION TRUE
//exemplar attribute DCM0 STARTUP_WAIT TRUE
//exemplar attribute DCM0 DFS_FREQUENCY_MODE LOW
//exemplar attribute DCM0 CLKFX_DIVIDE 1
//exemplar attribute DCM0 CLKFX_MULTIPLY 1
//exemplar attribute DCM0 CLK_FEEDBACK 1X
//exemplar attribute DCM0 CLKOUT_PHASE_SHIFT NONE
//exemplar attribute DCM0 PHASE_SHIFT 0

BUFG CLK_BUF0(
       .O(clock_out),
       .I(clk0)
       );

DCM DCM1 (
       .CLKFB(clock_with_ps_out),
       .CLKIN(clock),
       .DSSEN(low),
       .PSCLK(low),
       .PSEN(low),
       .PSINCDEC(low),
       .RST(low),
       .CLK0(clk1),
       .CLK90(),
       .CLK180(),
       .CLK270(),
       .CLK2X(),
       .CLK2X180(),
       .CLKDV(),
       .CLKFX(),
       .CLKFX180(),
       .LOCKED(dcm1_locked),
       .PSDONE(),
       .STATUS()
       );
```

```
//exemplar attribute DCM1 DLL_FREQUENCY_MODE LOW
//exemplar attribute DCM1 DUTY_CYCLE_CORRECTION TRUE
//exemplar attribute DCM1 STARTUP_WAIT TRUE
//exemplar attribute DCM1 DFS_FREQUENCY_MODE LOW
//exemplar attribute DCM1 CLKFX_DIVIDE 1
//exemplar attribute DCM1 CLKFX_MULTIPLY 1
//exemplar attribute DCM1 CLK_FEEDBACK 1X
//exemplar attribute DCM1 CLKOUT_PHASE_SHIFT FIXED
//exemplar attribute DCM1 PHASE_SHIFT 0

BUFG CLK_BUF1(
       .O(clock_with_ps_out),
       .I(clk1)
       );

// The following Verilog code is for simulation only
//exemplar translate_off
defparam DCM0.DLL_FREQUENCY_MODE = "LOW";
defparam DCM0.DUTY_CYCLE_CORRECTION = "TRUE";
defparam DCM0.STARTUP_WAIT = "TRUE";
defparam DCM0.DFS_FREQUENCY_MODE = "LOW";
defparam DCM0.CLKFX_DIVIDE = 1;
defparam DCM0.CLKFX_MULTIPLY = 1;
defparam DCM0.CLK_FEEDBACK = "1X";
defparam DCM0.CLKOUT_PHASE_SHIFT = "NONE";
defparam DCM0.PHASE_SHIFT = "0";
defparam DCM1.DLL_FREQUENCY_MODE = "LOW";
defparam DCM1.DUTY_CYCLE_CORRECTION = "TRUE";
defparam DCM1.STARTUP_WAIT = "TRUE";
defparam DCM1.DFS_FREQUENCY_MODE = "LOW";
defparam DCM1.CLKFX_DIVIDE = 1;
defparam DCM1.CLKFX_MULTIPLY = 1;
defparam DCM1.CLK_FEEDBACK = "1X";
defparam DCM1.CLKOUT_PHASE_SHIFT = "FIXED";
defparam DCM1.PHASE_SHIFT = "0";
//exemplar translate_on
endmodule // DCM_TOP
```

- Verilog example for FPGA Compiler II™

```
module DCM_TOP(
       clock_in,
       clock_out,
       clock_with_ps_out,
       reset
       );

input clock_in;
output clock_out;
output clock_with_ps_out;
output reset;

wire low;
wire high;
wire dcm0_locked;
wire dcm1_locked;
wire reset;
wire clk0;
wire clk1;
```

```
assign low = 1’b0;
assign high = 1’b1;
assign reset = !(dcm0_locked & dcm1_locked);

IBUFG CLOCK_IN (
       .I(clock_in),
       .O(clock)
       );

DCM DCM0 (
       .CLKFB(clock_out),
       .CLKIN(clock),
       .DSSEN(low),
       .PSCLK(low),
       .PSEN(low),
       .PSINCDEC(low),
       .RST(low),
       .CLK0(clk0),
       .CLK90(),
       .CLK180(),
       .CLK270(),
       .CLK2X(),
       .CLK2X180(),
       .CLKDV(),
       .CLKFX(),
       .CLKFX180(),
       .LOCKED(dcm0_locked),
       .PSDONE(),
       .STATUS());
/*synopsys attribute DLL_FREQUENCY_MODE “LOW” DUTY_CYCLE_CORRECTION
“TRUE” STARTUP_WAIT “TRUE” DFS_FREQUENCY_MODE “LOW” CLKFX_DIVIDE  “1”
CLKFX_MULTIPLY “1” CLK_FEEDBACK “1X” CLKOUT_PHASE_SHIFT “NONE”
PHASE_SHIFT  “0” */

BUFG CLK_BUF0(
       .O(clock_out),
       .I(clk0));

DCM DCM1 (
       .CLKFB(clock_with_ps_out),
       .CLKIN(clock),
       .DSSEN(low),
       .PSCLK(low),
       .PSEN(low),
       .PSINCDEC(low),
       .RST(low),
       .CLK0(clk1),
       .CLK90(),
       .CLK180(),
       .CLK270(),
       .CLK2X(),
       .CLK2X180(),
       .CLKDV(),
       .CLKFX(),
       .CLKFX180(),
       .LOCKED(dcm1_locked),
       .PSDONE(),
       .STATUS()
       );
```

```
/* synopsys attribute DLL_FREQUENCY_MODE “LOW” DUTY_CYCLE_CORRECTION
“TRUE” STARTUP_WAIT “TRUE” DFS_FREQUENCY_MODE “LOW” CLKFX_DIVIDE “1”
CLKFX_MULTIPLY “1” CLK_FEEDBACK “1X” CLKOUT_PHASE_SHIFT “FIXED”
PHASE_SHIFT “0” */

BUFG CLK_BUF1(
       .O(clock_with_ps_out),
       .I(clk1)
       );

// The following Verilog code is for simulation only
//synopsys translate_off
defparam DCM0.DLL_FREQUENCY_MODE = "LOW";
defparam DCM0.DUTY_CYCLE_CORRECTION = "TRUE";
defparam DCM0.STARTUP_WAIT = "TRUE";
defparam DCM0.DFS_FREQUENCY_MODE = "LOW";
defparam DCM0.CLKFX_DIVIDE = 1;
defparam DCM0.CLKFX_MULTIPLY = 1;
defparam DCM0.CLK_FEEDBACK = "1X";
defparam DCM0.CLKOUT_PHASE_SHIFT = "NONE";
defparam DCM0.PHASE_SHIFT = "0";

defparam DCM1.DLL_FREQUENCY_MODE = "LOW";
defparam DCM1.DUTY_CYCLE_CORRECTION = "TRUE";
defparam DCM1.STARTUP_WAIT = "TRUE";
defparam DCM1.DFS_FREQUENCY_MODE = "LOW";
defparam DCM1.CLKFX_DIVIDE = 1;
defparam DCM1.CLKFX_MULTIPLY = 1;
defparam DCM1.CLK_FEEDBACK = "1X";
defparam DCM1.CLKOUT_PHASE_SHIFT = "FIXED";
defparam DCM1.PHASE_SHIFT = "0";
//synopsys translate_on

endmodule // DCM_TOP
```

- Verilog example for XST

```
module DCM_TOP(
       clock_in,
       clock_out,
       clock_with_ps_out,
       reset
       );

input clock_in;
output clock_out;
output clock_with_ps_out;
output reset;

wire low;
wirehigh;
wire dcm0_locked;
wire dcm1_locked;
wire reset;
wire clk0;
wire clk1;
```

```
assign low = 1’b0;
assign high = 1’b1;
assign reset = (dcm0_locked ~& dcm1_locked);

IBUFG CLOCK_IN (
       .I(clock_in),
       .O(clock)
       );

DCM DCM0 (
       .CLKFB(clock_out),
       .CLKIN(clock),
       .DSSEN(low),
       .PSCLK(low),
       .PSEN(low),
       .PSINCDEC(low),
       .RST(low),
       .CLK0(clk0),
       .CLK90(),
       .CLK180(),
       .CLK270(),
       .CLK2X(),
       .CLK2X180(),
       .CLKDV(),
       .CLKFX(),
       .CLKFX180(),
       .LOCKED(dcm0_locked),
       .PSDONE(),
       .STATUS()
       );

BUFG CLK_BUF0(
       .O(clock_out),
       .I(clk0)
       );
// synthesis attribute DLL_FREQUENCY_MODE of DCM0 is "LOW"
// synthesis attribute DUTY_CYCLE_CORRECTION of DCM0 is "TRUE"
// synthesis attribute STARTUP_WAIT of DCM0 is "TRUE"
// synthesis attribute DFS_FREQUENCY_MODE of DCM0 is "LOW"
// synthesis attribute CLKFX_DIVIDE of DCM0 is "1"
// synthesis attribute CLKFX_MULTIPLY of DCM0 is "1"
// synthesis attribute CLK_FEEDBACK of DCM0 is "1X"
// synthesis attribute CLKOUT_PHASE_SHIFT of DCM0 is "FIXED"
// synthesis attribute PHASE_SHIFT of DCM0 is "0"
```

```
DCM DCM1 (
       .CLKFB(clock_with_ps_out),
       .CLKIN(clock),
       .DSSEN(low),
       .PSCLK(low),
       .PSEN(low),
       .PSINCDEC(low),
       .RST(low),
       .CLK0(clk1),
       .CLK90(),
       .CLK180(),
       .CLK270(),
       .CLK2X(),
       .CLK2X180(),
       .CLKDV(),
       .CLKFX(),
       .CLKFX180(),
       .LOCKED(dcm1_locked),
       .PSDONE(),
       .STATUS()
       );
// synthesis attribute DLL_FREQUENCY_MODE of DCM1 is "LOW"
// synthesis attribute DUTY_CYCLE_CORRECTION of DCM1 is "TRUE"
// synthesis attribute STARTUP_WAIT of DCM1 is "TRUE"
// synthesis attribute DFS_FREQUENCY_MODE of DCM1 is "LOW"
// synthesis attribute CLKFX_DIVIDE of DCM1 is "1"
// synthesis attribute CLKFX_MULTIPLY of DCM1 is "1"
// synthesis attribute CLK_FEEDBACK of DCM1 is "1X"
// synthesis attribute CLKOUT_PHASE_SHIFT of DCM1 is "FIXED"
// synthesis attribute PHASE_SHIFT of DCM1 is "0"

BUFG CLK_BUF1(
       .O(clock_with_ps_out),
       .I(clk1));

// The following Verilog code is for simulation only
//synthesis translate_off
defparam DCM0.DLL_FREQUENCY_MODE = "LOW";
defparam DCM0.DUTY_CYCLE_CORRECTION = "TRUE";
defparam DCM0.STARTUP_WAIT = "TRUE";
defparam DCM0.DFS_FREQUENCY_MODE = "LOW";
defparam DCM0.CLKFX_DIVIDE = 1;
defparam DCM0.CLKFX_MULTIPLY = 1;
defparam DCM0.CLK_FEEDBACK = "1X";
defparam DCM0.CLKOUT_PHASE_SHIFT = "NONE";
defparam DCM0.PHASE_SHIFT = "0";

defparam DCM1.DLL_FREQUENCY_MODE = "LOW";
defparam DCM1.DUTY_CYCLE_CORRECTION = "TRUE";
defparam DCM1.STARTUP_WAIT = "TRUE";
defparam DCM1.DFS_FREQUENCY_MODE = "LOW";
defparam DCM1.CLKFX_DIVIDE = 1;
defparam DCM1.CLKFX_MULTIPLY = 1;
defparam DCM1.CLK_FEEDBACK = "1X";
defparam DCM1.CLKOUT_PHASE_SHIFT = "FIXED";
defparam DCM1.PHASE_SHIFT = "0";
//synthesis translate_on

endmodule // DCM_TOP
```

# Using Dedicated Global Set/Reset Resource

The use of Global Set / Reset Resource (GSR) in Virtex™, Virtex-E™, Virtex-II™ and Spartan-II™ devices must be considered carefully. Synthesis tools do not automatically infer GSRs for these devices; however, STARTUP_VIRTEX, STARTUP_VIRTEX2 and STARTUP_SPARTAN2 can be instantiated in your HDL code to access the GSR resources. Xilinx® recommends that for Virtex™, Virtex-E™, and Spartan-II™ designs, you write the high fanout set / reset signal explicitly in the HDL code, and not use the STARTUP_VIRTEX, STARTUP_VIRTEX2 or STARTUP_SPARTAN2 blocks. There are two advantages to this method.

1. This method gives you a faster speed. The set / reset signals are routed onto the secondary longlines in the device, which are global lines with minimal skews and high speed. Therefore, the reset / set signals on the secondary lines are much faster than the GSR nets of the STARTUP_VIRTEX block. Since Virtex™ is rich in routings, placing and routing this signal on the global lines can be easily done by our software.
2. The **`trce`** program analyzes the delays of the explicitly written set / reset signals. You can read the TWR file (report file of the **`trce`** program) and find out exactly how fast your design’s speed is. The **`trce`** program does not analyze the delays on the GSR net of the STARTUP_VIRTEX, STARTUP_VIRTEX2, or STARTUP_SPARTAN2. Hence, using an explicit set / reset signal improves your design accountability.

For Virtex™ / E / II / II Pro / II Pro X and Spartan™-II / 3 devices, the Global Set / Reset (GSR) signal is, by default, set to active high (globally resets device when logic equals 1). You can change this to active low by inverting the GSR signal before connecting it to the GSR input of the STARTUP component.

***Note:*** See Chapter 6, “Verifying Your Design” for more information on simulating the Global Set/Reset.

## Startup State

The GSR pin on the STARTUP block or the GSRIN pin on the STARTBUF block drives the GSR net and connects to each flip-flop’s Preset and Clear pin. When you connect a signal from a pad to the STARTUP block’s GSR pin, the GSR net is activated. Because the GSR net is built into the silicon it does not appear in the pre-routed netlist file. When the GSR signal is asserted High (the default), all flip-flops and latches are set to the state they were in at the end of configuration. When you simulate the routed design, the gate simulator translation program correctly models the GSR function.

See Chapter 6, “Verifying Your Design” for more information on STARTUP and STARTBUF.

***Note:*** The following VHDL and Verilog example shows a STARTUP_VIRTEX instantiation using both GSR and GTS pins for FPGA Compiler II™, LeonardoSpectrum™ and XST.

- VHDL example

```
-- This example uses both GTS and GSR pins.
-- Unconnected STARTUP pins are omitted from
-- component declaration.
library IEEE;
use IEEE.std_logic_1164.all;
entity setreset is
    port (CLK: in std_logic;
        DIN1 : in STD_LOGIC;
        DIN2 : in STD_LOGIC;
        RESET : in STD_LOGIC;
        GTSInput : in STD_LOGIC;
        DOUT1 : out STD_LOGIC;
        DOUT2 : out STD_LOGIC;
        DOUT3 : out STD_LOGIC);
end setreset ;
architecture RTL of setreset is
component STARTUP_VIRTEX
      port( GSR, GTS: in std_logic);
end component;
begin
startup_inst: STARTUP_VIRTEX port map(GSR => RESET, GTS => GTSInput);
reset_process: process (CLK, RESET)
      begin
        if (RESET = ’1’) then
          DOUT1 <= ’0’;
        elsif ( CLK’event and CLK =’1’) then
        DOUT1 <= DIN1;
        end if;
end process;
gtsprocess:process (GTSInput)
begin
  if GTSInput = ’0’ then
    DOUT3 <= ’0’;
    DOUT2 <= DIN2;
  else
    DOUT2 <= ’Z’;
    DOUT3 <= ’Z’;
  end if;
end process;
end RTL;
```

- Verilog example

```
// This example uses both GTS and GSR pins
// Unused STARTUP pins are omitted from module
// declaration.
module setreset(CLK,DIN1,DIN2,RESET,GTSInput,DOUT1,DOUT2,DOUT3);
input CLK;
input DIN1;
input DIN2;
input RESET;
input GTSInput;
output DOUT1;
output DOUT2;
output DOUT3;
reg DOUT1;
STARTUP_VIRTEX startup_inst( .GSR(RESET), .GTS(GTSInput));
always @(posedge CLK or posedge RESET)
begin
  if (RESET)
        DOUT1 <= 1'b0;
  else
        DOUT1 <= DIN1;
end
assign DOUT3 = (GTSInput == 1'b0)? 1'b0: 1'bZ;
assign DOUT2 = (GTSInput == 1'b0)? DIN2: 1'bZ;
endmodule
```

The following VHDL / Verilog examples show a STARTUP_VIRTEX instantiation using both GSR and GTS pins in Synplify™. In the examples, STARTUP_VIRTEX_GSR and STARTUP_VIRTEX_GTS are instantiated together to get the GSR and GTS pins connected. The resulting EDIF netlist has only one STARTUP_VIRTEX block with GTS and GSR connections. The CLK pin of the STARTUP_VIRTEX are unconnected. If all pins (GSR, GTS, and CLK) in the STARTUP block are needed, use STARTUP_VIRTEX to port map the pins.

- VHDL example

```
library IEEE,virtex,synplify;
use synplify.attributes.all;
use virtex.components.all;
use IEEE.std_logic_1164.all;
entity setreset is
  port (CLK: in std_logic;
        DIN1 : in STD_LOGIC;
        DIN2 : in STD_LOGIC;
        RESET : in STD_LOGIC;
        GTSInput : in STD_LOGIC;
        DOUT1 : out STD_LOGIC;
        DOUT2 : out STD_LOGIC;
        DOUT3 : out STD_LOGIC
        );
end setreset ;
architecture RTL of setreset is
begin
  u0: STARTUP_VIRTEX_GSR port map(GSR => RESET);
  u1: STARTUP_VIRTEX_GTS port map(GTS => GTSInput);
  reset_process: process (CLK, RESET)
      begin
        if (RESET = ’1’) then
          DOUT1 <= ’0’;
        elsif ( CLK’event and CLK =’1’) then
          DOUT1 <= DIN1;
        end if;
end process;
gtsprocess:process (GTSInput)
begin
  if GTSInput = ’0’ then
      DOUT3 <= ’0’;
      DOUT2 <= DIN2;
  else
      DOUT2 <= ’Z’;
      DOUT3 <= ’Z’;
  end if;
end process;
end RTL;
```

- Verilog example

```
'include "path/to/virtex.v"
module setreset(CLK,DIN1,DIN2,RESET,GTSInput,DOUT1,DOUT2,DOUT3);
  input CLK;
  input DIN1;
  input DIN2;
  input RESET;
  input GTSInput;
  output DOUT1;
  output DOUT2;
  output DOUT3;
  reg DOUT1;
  STARTUP_VIRTEX_GSR startup_inst(.GSR(RESET));
  STARTUP_VIRTEX_GTS startup_2(.GTS(GTSInput));
  always @(posedge CLK or posedge RESET)
    begin
      if (RESET)
          DOUT1 <= 1'b0;
      else
          DOUT1 <= DIN1;
    end
  assign DOUT3 = (GTSInput == 1'b0)? 1'b0: 1'bZ;
  assign DOUT2 = (GTSInput == 1'b0)? DIN2: 1'bZ;
endmodule
```

## Preset vs. Clear

The Virtex™, Virtex-E™, Virtex-II™, Virtex-II Pro™, Virtex-II Pro X™ and Spartan-II™, Spartan-3™ family flip-flops are configured as either preset (asynchronous set) or clear (asynchronous reset) during startup. Automatic assertion of the GSR net presets or clears each flip-flop after the FPGA is configured. You can assert the GSR pin at any time to produce this global effect. You can also preset or clear individual flip-flops with the flip-flop's dedicated Preset or Clear pin. When a Preset or Clear pin on a flip-flop is connected to an active signal, the state of that signal controls the startup state of the flip-flop. For example, if you connect an active signal to the Preset pin, the flip-flop starts up in the preset state. If you do not connect the Clear or Preset pin, the default startup state is a clear state. To change the default to preset, assign an INIT=1 to the Virtex™/E/II/II Pro/ II Pro X or Spartan™-II/3 flip-flop.

I/O flip-flops and latches in Virtex™, Virtex-E™, Virtex-II™, Virtex-II Pro™, Virtex-II Pro X™ and Spartan™-II/3 have an SR pin which can be configured as a synchronous Set, a synchronous Reset, an asynchronous Preset, or an asynchronous Clear. The SR pin can be driven by any user logic, but INIT also works for these flip-flops.

Following are examples of setting register INIT using ROCBUF. In the HDL code, the instantiated ROCBUF connects the set/reset signal. The Xilinx® tools automatically remove the ROCBUF during implementation leaving the set/reset signal active only during power-up.

- VHDL example

```
library IEEE;
 use IEEE.std_logic_1164.all;
  entity d_register is
    port (CLK : in std_logic;
      RESET : in std_logic;
      D0 : in std_logic;
      D1 : in std_logic;
      Q0 : out std_logic;
      Q1 : out std_logic);
  end d_register;
architecture XILINX of d_register is
signal RESET_int : std_logic;
component ROCBUF is port (
        I : in STD_LOGIC;
        O : out STD_LOGIC
        );
end component;
begin
  U1: ROCBUF port map (
        I => RESET,
        O => RESET_int
        );
  process (CLK, RESET_int)
  begin
    if RESET_int = ’1’ then
        Q0 <= ’0’;
        Q1 <= ’1’;
    elsif rising_edge(CLK) then
        Q0 <= D0;
        Q1 <= D1;
    end if;
  end process;
end XILINX;
```

- Verilog example

```
/* Note: In Synplify, set blackbox attribute for ROCBUF as follows:
     module ROCBUF(I, O);      //synthesis syn_black_box
     input I;
     output O;
     endmodule
*/
module ROCBUF (I, O);
  input I;
  output O;
endmodule
module rocbuf_example (reset, clk, d0, d1, q0, q1);
  input reset;
  input clk;
  input d0;
  input d1;
  output q0;
  output q1;
  reg q0, q1;
  wire reset_int;
  ROCBUF u1 ( .I(reset), .O(reset_int));
  always @ (posedge clk or posedge reset_int)
    begin
      if (reset_int)
        begin
          q0 <= 1’b0;
          q1 <= 1’b1;
        end
      else
        begin
          q0 <= d0;
          q1 <= d1;
        end
    end
endmodule
```

## Implementing Inputs and Outputs

FPGAs have limited logic resources in the user-configurable input/output blocks (IOB). You can move logic that is normally implemented with CLBs to IOBs. By moving logic from CLBs to IOBs, additional logic can be implemented in the available CLBs. Using IOBs also improves design performance by increasing the number of available routing resources.

The Virtex™, Virtex-E™, Virtex-II™, Virtex-II Pro™, Virtex-II Pro X™, Spartan-II™ and Spartan-3™ IOBs feature SelectIO™ inputs and outputs that support a wide variety of I/O signaling standards. In addition, each IOB provides three storage elements. The following sections discuss IOB features in more detail.

## I/O Standards

The following table summarizes the I/O standards supported in Virtex™/E/II/II Pro/II Pro X and Spartan™-II/3 devices. A complete table is available in the *Libraries Guide*.

*Table 4-3:* **I/O Standard in Virtex™/E/II and Spartan-II™ Devices**

| I/O Standard | Virtex™/ Spartan-II™ | Virtex-E™ | Virtex™-II/II Pro/ II Pro X/ Spartan-3™ |
|---|---|---|---|
| LVTTL (default) | ÷ | ÷ | ÷ |
| AGP | ÷ | ÷ | ÷ |
| CTT | ÷ | ÷ | |
| GTL | ÷ | ÷ | ÷ |
| GTLP | ÷ | ÷ | ÷ |
| HSTL Class I | ÷ | ÷ | ÷ |
| HSTL Class II | | | ÷ |
| HSTL Class III | ÷ | ÷ | ÷ |
| HSTL Class IV | ÷ | ÷ | ÷ |
| LVCMOS2 | ÷ | | |
| LVCMOS15 | | | ÷ |
| LVCMOS18 | | ÷ | ÷ |
| LVCMOS25, 33 | | | ÷ |
| LVCZ_15, 18, 25, 33 | | | ÷ |
| LVCZ_DV2_15, 18, 25, 33 | | | ÷ |
| LVDS | | ÷ | ÷ |
| LVPECL | | ÷ | ÷ |
| PCI33_5 | | | |
| PCI33_3, PCI66_3 | ÷ | ÷ | ÷ |
| PCIX | | | ÷ |

*Table 4-3:* **I/O Standard in Virtex™/E/II and Spartan-II™ Devices**

| I/O Standard | Virtex™/ Spartan-II™ | Virtex-E™ | Virtex™-II/II Pro/ II Pro X/ Spartan-3™ |
|---|---|---|---|
| SSTL2 Class I and Class II | ÷ | ÷ | ÷ |
| SSTL3 Class I and Class II | ÷ | ÷ | ÷ |

For Virtex™, Virtex-E™, and Spartan-II™ devices, Xilinx® provides a set of IBUF, IBUFG, IOBUF, and OBUF with its SelectIO™ variants. For example, IBUF_GTL, IBUFG_PCI66_3, IOBUF_HSTL_IV, OBUF_LVCMOS2. Alternatively, an IOSTANDARD attribute can be set to a specific I/O standard and attached to an IBUF, IBUFG, IOBUF, and OBUF. The IOSTANDARD attribute can be set in the user constraint file (UCF) or in the netlist by the synthesis tool.

The Virtex-II™ library includes certain SelectIO™ components for compatibility with other architectures. However, the recommended method for using SelectIO™ components for Virtex-II™ is to attach an IOSTANDARD attribute to IBUF/IBUFG/IOBUF/OBUF. For example, attach IOSTANDARD=GTLP to an IBUF instead of using the IBUF_GTLP.

The default for the IOSTANDARD attribute is LVTTL. For all Virtex™/E/II and Spartan-II™ devices, you must specify IBUF, IBUFG, IOBUF or OBUF on the IOSTANDARD attribute if LVTTL is not desired.

For more information on I/O standards and components, please refer to the *Libraries Guide*.

## Inputs

Virtex™, Virtex-E™, Virtex-II™, Virtex-II Pro™, Virtex-II Pro X™, Spartan-II™, and Spartan-3™ inputs can be configured to the I/O standards listed above.

In FPGA Compiler II™, these special IOB components exist in the synthesis library and can be instantiated in your HDL code or selected from the FPGA Compiler II™ constraints GUI. A complete list of components understood by FPGA Compiler II™ can be found in the `lib\virtex` directory under the FPGA Compiler II™ tree (`%XILINX%\synth` for ISE users). FPGA Compiler II™ understands these components and does not attempt to place any I/O logic on these ports. Users are alerted by this warning:

```
Warning: Existing pad cell '/ver1-Optimized/U1' is connected to the
port 'clk' - no pads cells inserted at this port. (FPGA-PADMAP-1)
```

In LeonardoSpectrum™, insert appropriate buffers on selected ports in the constraints editor. Alternatively, you can set the following attribute in TCL script after the **`read`** but before the **`optimize`** options.

**`PAD`** *IOstandard portname*

The following is an example of setting an I/O standard in LeonardoSpectrum™.

```
PAD IBUF_AGP data (7:0)
```

In Synplify™, users can set XC_PADTYPE attribute in SCOPE (Synplify's constraint editor) or in HDL code as shown in the following example:

- VHDL example

```
library ieee, synplify;
  use ieee.std_logic_1164.all;
  use synplify.attributes.all;
  entity test_padtype is
    port( a : in std_logic_vector(3 downto 0);
      b : in std_logic_vector(3 downto 0);
      clk, rst, en : in std_logic;
      bidir : inout std_logic_vector(3 downto 0);
      q : out std_logic_vector(3 downto 0));
  attribute xc_padtype of a : signal is "IBUF_SSTL3_I";
  attribute xc_padtype of bidir : signal is "IOBUF_HSTL_III";
  attribute xc_padtype of q : signal is "OBUF_S_8";
  end entity;
```

- Verilog example

```
module test_padtype (a, b, clk, rst, en, bidir, q);
input [3:0] a /* synthesis xc_padtype = "IBUF_AGP" */;
input [3:0] b;
input clk, rst, en;
inout [3:0] bidir /* synthesis xc_padtype = "IOBUF_CTT" */;
output [3:0] q /* synthesis xc_padtype = "OBUF_F_12" */;
```

***Note:*** Refer to IBUF_selectIO in the *Libraries Guide* for a list of available IBUF_selectIO values.

## Outputs

Virtex™/E/II/II Pro/II Pro X and Spartan™-II/3 outputs can also be configured to any of I/O standards listed in the I/O standards section. An OBUF that uses the LVTTL, LVCMOS15, LVCMOS18, LVCMOS25, or LVCMOS33 signaling standards has selectable drive and slew rates using the DRIVE and SLOW or FAST constraints. The defaults are DRIVE=12 mA and SLOW slew.

In addition, you can control the slew rate and drive power for LVTTL outputs using **OBUF**_*slew_drivepower.*

Refer to OBUF_selectIO in the *Libraries Guide* for a list of available OBUF_selectIO values. You can use the examples in the Inputs section to configure OBUF to an I/O standard.

## Using IOB Register and Latch

Virtex™, Virtex-E™, and Spartan-II™ IOBs contain three storage elements. The three IOB storage elements function either as edge-triggered D-type flip-flops or as level sensitive latches. Each IOB has a clock (CLK) signal shared by the three flip-flops and independent clock enable (CE) signals for each flip-flop.

In addition to the CLK and CE control signals, the three flip-flops share a Set/Reset (SR). However, each flip-flop can be independently configured as a synchronous set, a synchronous reset, an asynchronous preset, or an asynchronous clear. FDCP (asynchronous reset and set) and FDRS (synchronous reset and set) register configurations are not available in IOBs.

Virtex™-II/II Pro/II Pro X or Spartan-3™ IOBs also contain three storage elements with an option to configure them as FDCP, FDRS, and Dual-Data Rate (DDR) registers. Each register has an independent CE signal. The OTCLK1 and OTCLK2 clock pins are shared between the output and tristate enable register. A separate clock (ICLK1 and ICLK2) drives the input register. The set and reset signals (SR and REV) are shared by the three registers.

Virtex™, Virtex-E™, Virtex-II™, Virtex-II Pro™, Virtex-II Pro X™, Spartan-II™ and Spartan-3™ devices no longer have primitives that correspond to the synchronous elements in the IOBs. There are a few ways to infer usage of these flip-flops if the rules for pulling them into the IOB are followed. The following rules apply.

- All flip-flops that are to be pulled into the IOB must have a fanout of 1. This applies to output and tristate enable registers. For example, if there is a 32 bit bidirectional bus, then the tristate enable signal must be replicated in the original design so that it has a fanout of 1.
- In Virtex™/E and Spartan-II™ devices, all flip-flops must share the same clock and reset signal. They can have independent clock enables.
- In Virtex™-II/II Pro/II Pro X or Spartan-3™ devices, output and tristate enable registers must share the same clock. All flip-flops must share the same set and reset signals.

One way you can pull flip-flops into the IOB is to use the IOB=TRUE setting. Another way is to pull flip-flops into the IOB using the **map —pr** command, which is discussed in a later section. Some synthesis tools apply the IOB=TRUE attribute and allow you to merge a flip-flop to an IOB by setting an attribute. Refer to your synthesis tool documentation for the correct attribute and settings.

In FPGA Compiler II™, you can set the attribute through the FPGA Compiler II™ constraints editor for each port into which a flip-flop should be merged. For tristate enable flip-flops, set the default value for 'Use I/O Reg' to TRUE. This causes the IOB=TRUE constraint to be written on every flip-flop in the design.

LeonardoSpectrum™, through ISE, can push registers into IOBs. Right click on the **Synthesize** process, click **Properties**, click the Architecture Options tab and enable the **Map to IOB Registers** setting.

In standalone LeonardoSpectrum™, you can select **Map IOB Registers** from the Technology tab in the GUI or set the following attribute in your TCL script:

```
set virtex_map_iob_registers TRUE
```

In Synplify™, attach the SYN_USEIOFF attribute to the module or architecture of the top-level in one of these ways:

- Add the attribute in SCOPE. The constraint file syntax looks like this:

  ```
  define_global_attribute syn_useioff 1
  ```

- Add the attribute in the VHDL/Verilog top-level source code as follows:
  - ♦ VHDL example

    ```
    architecture rtl of test is
      attribute syn_useioff : boolean;
      attribute syn_useioff of rtl : architecture is true;
    ```

  - ♦ Verilog example

    ```
    module test(d, clk, q) /* synthesis syn_useioff = 1 */;
    ```

## Using Dual Data Rate IOB Registers

The following VHDL and Verilog examples show how to infer dual data rate registers for inputs only. See the Using IOB Register and Latch section for an attribute to enable I/O register inference in your synthesis tool. The dual data rate register primitives (the synchronous set/reset with clock enable FDDRRSE, and asynchronous set/reset with clock enable FDDRCPE) must be instantiated in order to utilize the dual data rate registers in the outputs. Please refer to the Instantiating Components section for information on instantiating primitives.

- VHDL example

```
library ieee;
use ieee.std_logic_1164.all;
entity ddr_input is
  port (
      clk : in std_logic;
      d   : in std_logic;
      rst : in std_logic;
      q1  : out std_logic;
      q2  : out std_logic
      );
end ddr_input;
architecture behavioral of ddr_input is
begin
  q1reg : process (clk, rst)
  begin
    if rst = ’1’ then
      q1 <= ’0’;
    elsif clk’event and clk=’1’ then
      q1 <= d;
    end if;
  end process;
  q2reg : process (clk, rst)
  begin
    if rst = ’1’ then
      q2 <= ’0’;
    elsif clk’event and clk=’0’ then
      q2 <= d;
    end if;
  end process;
end behavioral;
```

- Verilog example

```
module ddr_input (data_in, data_out, clk, rst);
input data_in, clk, rst;
output data_out;
reg q1, q2;
always @ (posedge clk or posedge rst)
  begin
    if (rst)
      q1 <=1’b0;
    else
      q1 <= data_in;
  end
always @ (negedge clk or posedge rst)
  begin
    if (rst)
      q2 <=1’b0;
    else
      q2 <= data_in;
  end
assign data_out = q1 & q2;
end module
```

## Using Output Enable IOB Register

The following VHDL and Verilog examples illustrate how to infer an output enable register. See the above section for an attribute to turn on I/O register inference in synthesis tools.

***Note:*** If using FPGA Compiler II™ to synthesize the following examples, open up FPGA Compiler II's constraints editor, select the Ports tab and change the default Use I/O Reg option from NONE to TRUE. Doing so places an IOB=TRUE constraint on every flip-flop in the design. There is no option to specify only the output enable registers.

- VHDL example

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity tri_state is
Port (
        data_in_p : in std_logic_vector(7 downto 0);
        clk : in std_logic;
        tri_state_a : in std_logic;
        tri_state_b : in std_logic;
        data_out : out std_logic_vector(7 downto 0)
        );
end tri_state;
architecture behavioral of tri_state is
signal data_in : std_logic_vector(7 downto 0);
signal data_in_r : std_logic_vector(7 downto 0);
signal tri_state_cntrl : std_logic_vector(7 downto 0);
signal temp_tri_state : std_logic_vector(7 downto 0);
begin
  G1: for I in 0 to 7 generate
    temp_tri_state(I) <= tri_state_a AND
    tri_state_b;   -- create duplicate input signal
  end generate;
  process (tri_state_cntrl, data_in_r)
  begin
    G2:  for J in 0 to 7 loop
      if (tri_state_cntrl(J) = ’0’) then  -- tri-state data_out
          data_out(J) <= data_in_r(J);
      else
          data_out(J) <= ’Z’;
      end if;
    end loop;
  end process;
  process(clk) begin
    if clk’event and clk=’1’ then
      data_in <= data_in_p;    -- register for input
      data_in_r <= data_in;    -- register for output
      for I in 0 to 7 loop
        tri_state_cntrl(I) <= temp_tri_state(I);
        -- register tri-state
      end loop;
    end if;
  end process;
end behavioral;
```

- Verilog example

```
/////////////////////////////////////////////////
// Inferring output enable register            //
/////////////////////////////////////////////////
module tri_state (data_in_p,clk,tri_state_a,tri_state_b, data_out);
  input[7:0] data_in_p;
  input clk;
  input tri_state_a;
  input tri_state_b;
  output[7:0] data_out;
  reg[7:0] data_out;
  reg[7:0] data_in;
  reg[7:0] data_in_r;
  reg[7:0] tri_state_cntrl;
  wire[7:0] temp_tri_state;
// create duplicate input signal
  assign temp_tri_state[0] = tri_state_a & tri_state_b;
  assign temp_tri_state[1] = tri_state_a & tri_state_b;
  assign temp_tri_state[2] = tri_state_a & tri_state_b;
  assign temp_tri_state[3] = tri_state_a & tri_state_b;
  assign temp_tri_state[4] = tri_state_a & tri_state_b;
  assign temp_tri_state[5] = tri_state_a & tri_state_b;
  assign temp_tri_state[6] = tri_state_a & tri_state_b;
  assign temp_tri_state[7] = tri_state_a & tri_state_b;
// exemplar attribute temp_tri_state
  preserve_signal TRUE
  always @(tri_state_cntrl or data_in_r)
    begin
      begin : xhdl_1
        integer J;
        for (J = 0; J <= 7; J = J + 1)
        begin : G2
          if (!(tri_state_cntrl[J]))
            begin
              data_out[J] <= data_in_r[J];
            end
          else    //tri-state data_out
            begin
              data_out[J] <= 1’bz;
            end
        end
      end
    end
  always @(posedge clk)
    begin
      data_in <= data_in_p;    // register for input
      data_in_r <= data_in;    // register for output
      tri_state_cntrl[0] <= temp_tri_state[0] ;
      tri_state_cntrl[1] <= temp_tri_state[1] ;
      tri_state_cntrl[2] <= temp_tri_state[2] ;
      tri_state_cntrl[3] <= temp_tri_state[3] ;
      tri_state_cntrl[4] <= temp_tri_state[4] ;
      tri_state_cntrl[5] <= temp_tri_state[5] ;
      tri_state_cntrl[6] <= temp_tri_state[6] ;
      tri_state_cntrl[7] <= temp_tri_state[7] ;
    end
endmodule
```

### Using -pr Option with Map

Use the –pr (pack registers) option when running Map. This option specifies to the Map program to move registers into IOBs when possible. Use the following is the syntax.

```
map –pr {i|o|b} input_file_name |output_file_name
```

Example:

```
map -pr b design_name.ngd
```

## Virtex-E™ IOBs

Virtex-E™ has the same IOB structure and features as Virtex™ and Spartan-II™ devices except for the available I/O standards.

### Additional I/O Standards

Virtex-E™ devices have two additional I/O standards: LVPECL and LVDS.

Because LVDS and LVPECL require two signal lines to transmit one data bit, it is handled differently from any other I/O standards. A UCF or an NGC file with complete pin LOC information must be created to ensure that the I/O banking rules are not violated. If a UCF or NGC file is not used, PAR issues errors.

The input buffer of these two I/O standards may be placed in a wide number of IOB locations. The exact locations are dependent on the package that is used. The Virtex-E™ package information lists the possible locations as IO_L#P for the P-side and IO_L#N for the N-side where # is the pair number. Only one input buffer is required to be instantiated in the design and placed on the correct IO_L#P location. The N-side of the buffer is reserved and no other IOB is allowed on this location.

The output buffer may be placed in a wide number of IOB locations. The exact locations are dependent on the package that is used. The Virtex-E™ package information lists the possible locations as IO_L#P for the P-side and IO_L#N for the N-side where # is the pair number. However, both output buffers are required to be instantiated in the design and placed on the correct IO_L#P and IO_L#N locations. In addition, the output (O) pins must be inverted with respect to each other. (one HIGH and one LOW). Failure to follow these rules leads to DRC errors in the software.

The following examples show VHDL and Verilog coding for LVDS I/O standards targeting a V50ECS144 device. An AUCF example is also provided.

- VHDL example

```
library IEEE;
use IEEE.std_logic_1164.all;
entity LVDSIO is
  port (
        CLK, DATA, Tin : in STD_LOGIC;
        IODATA_p, IODATA_n : inout STD_LOGIC;
        Q_p, Q_n : out STD_LOGIC
        );
end LVDSIO;
architecture BEHAV of LVDSIO is
  component IBUF_LVDS is port (
        I : in STD_LOGIC;
        O : out STD_LOGIC
        );
  end component;
```

```
component OBUF_LVDS is port (
     I : in STD_LOGIC;
     O : out STD_LOGIC
     );
end component;
component IOBUF_LVDS is port (
     I : in STD_LOGIC;
     T : in STD_LOGIC;
     IO : inout STD_LOGIC;
     O : out STD_LOGIC
     );
end component;
component INV is port (
     I : in STD_LOGIC;
     O : out STD_LOGIC
     );
end component;
component IBUFG_LVDS is port(
     I : in STD_LOGIC;
     O : out STD_LOGIC
     );
end component;
component BUFG is port(
     I : in STD_LOGIC;
     O : out STD_LOGIC
     );
end component;
signal iodata_in    : std_logic;
signal iodata_n_out : std_logic;
signal iodata_out   : std_logic;
signal DATA_int     : std_logic;
signal Q_p_int      : std_logic;
signal Q_n_int      : std_logic;
signal CLK_int      : std_logic;
signal CLK_ibufgout : std_logic;
signal Tin_int      : std_logic;
begin
  UI1: IBUF_LVDS port map (
     I => DATA,
     O => DATA_int
     );
  UI2: IBUF_LVDS port map (
     I => Tin,
     O => Tin_int
     );
  UO_p: OBUF_LVDS port map (
     I => Q_p_int,
     O => Q_p
     );
  UO_n: OBUF_LVDS port map (
     I => Q_n_int,
     O => Q_n
     );
  UIO_p: IOBUF_LVDS port map (
     I => iodata_out,
     T => Tin_int,IO => iodata_p,
     O => iodata_in
     );
```

```
    UIO_n: IOBUF_LVDS port map (
        I => iodata_n_out,
        T => Tin_int,
        IO => iodata_n,
        O => open
        );
    UINV: INV port map (
        I => iodata_out,
        O => iodata_n_out
        );
    UIBUFG: IBUFG_LVDS port map (
        I => CLK,
        O => CLK_ibufgout
        );
    UBUFG: BUFG port map (
        I => CLK_ibufgout,
        O => CLK_int
        );

  My_D_Reg: process (CLK_int, DATA_int)
  begin
        if (CLK_int’event and CLK_int=’1’) then
          Q_p_int <= DATA_int;
        end if;
    end process;        -- End My_D_Reg
  iodata_out <= DATA_int and iodata_in;
  Q_n_int <= not Q_p_int;
end BEHAV;
```

- Verilog example

```
module LVDSIOinst (CLK,DATA,Tin,IODATA_p,IODATA_n,Q_p,Q_n);
input CLK, DATA, Tin;
inout IODATA_p, IODATA_n;
output  Q_p, Q_n;

wire iodata_in;
wire iodata_n_out;
wire iodata_out;
wire DATA_int;
reg Q_p_int;
wire Q_n_int;
wire CLK_int;
wire CLK_ibufgout;
wire Tin_int;

IBUF_LVDS UI1 (.I(DATA), .O(DATA_int));
IBUF_LVDS UI2 (.I(Tin), .O (Tin_int));
OBUF_LVDS UO_p (.I(Q_p_int), .O(Q_p));
OBUF_LVDS UO_n (.I(Q_n_int), .O(Q_n));
IOBUF_LVDS UIO_p(
       .I(iodata_out),
       .T(Tin_int)
       .IO(IODATA_p),
       .O(iodata_in)
       );
```

```
IOBUF_LVDS UIO_n (
       .I (iodata_n_out),
       .T(Tin_int),
       .IO(IODATA_n), .O ()
       );
INV UINV ( .I(iodata_out), .O(iodata_n_out));
IBUFG_LVDS UIBUFG ( .I(CLK), .O(CLK_ibufgout));
BUFG UBUFG ( .I(CLK_ibufgout), .O(CLK_int));

always @ (posedge CLK_int)
  begin
    Q_p_int <= DATA_int;
  end
  assign iodata_out = DATA_int && iodata_in;
  assign Q_n_int = ~Q_p_int;
endmodule
```

- UCF example targeting V50ECS144

```
NET CLK LOC = A6;        #GCLK3
NET DATA LOC = A4;       #IO_L0P_YY
NET Q_p LOC = A5;        #IO_L1P_YY
NET Q_n LOC = B5;        #IO_L1N_YY
NET iodata_p LOC = D8;   #IO_L3P_yy
NET iodata_n LOC = C8;   #IO_L3N_yy
NET Tin LOC = F13;       #IO_L10P
```

The following examples use the IOSTANDARD attribute on I/O buffers as a work around for LVDS buffers. This example can also be used with other synthesis tools to configure I/O standards with the IOSTANDARD attribute.

- VHDL example

```
library IEEE;
use IEEE.std_logic_1164.all;
entity flip_flop is
  port(
       d : in std_logic;
       clk : in std_logic;
       q : out std_logic;
       q_n : out std_logic
       );
end flip_flop;

architecture flip_flop_arch of flip_flop is
  component IBUF
  port(
       I: in std_logic;
       O: out std_logic
       );
  end component;
  component OBUF
  port(
       I: in std_logic;
       O: out std_logic);
  end component;
  attribute IOSTANDARD : string;
  attribute LOC : string;
  attribute IOSTANDARD of u1 : label is "LVDS";
  attribute IOSTANDARD of u2 : label is "LVDS";
  attribute IOSTANDARD of u3 : label is "LVDS";
```

```
------------------------------------------------
-- Pin location A5 on the cs144
-- package represents the
-- ’positive’ LVDS pin.
-- Pin location D8 represents the
-- ’positive’ LVDS pin.
-- Pin location C8 represents the
-- ’negative’ LVDS pin.
------------------------------------------------
  attribute LOC of u1 : label is "A5";
  attribute LOC of u2 : label is "D8";
  attribute LOC of u3 : label is "C8";
  signal d_lvds, q_lvds, q_lvds_n : std_logic;
  begin
    u1 : IBUF port map (d,d_lvds);
    u2 : OBUF port map (q_lvds,q);
    u3 : OBUF port map (q_lvds_n,q_n);
    process (clk) begin
      if clk’event and clk = ’1’ then
        q_lvds <= d_lvds;
      end if;
    end process;
  q_lvds_n <= not(q_lvds);
end flip_flop_arch;
```

- Verilog example

```
module flip_flop (d, clk, q, q_n);
/*********************************/
// Pin location A5 on the Virtex-E
// cs144 package represents the
// ’positive’ LVDS pin.
// Pin location D8 represents the
// ’positive’ LVDS pin.
// Pin location C8 represents the
// ’negative’ LVDS pin.
/*********************************/
  input          /*synthesis xc_loc="A5"*/;
  input clk;
  output q       /*synthesis xc_loc="D8"*/;
  output q_n     /*synthesis xc_loc="C8"*/;

  //synthesis attribute LOC d "A5"
  //synthesis attribute LOC q "D8"
  //synthesis attribute LOC q_n "C8"

  //exemplar attribute d pin_number A5
  //exemplar attribute q pin_number D8
  //exemplar attribute q_n pin_number C8

  wire d, clk, d_lvds, q;
  reg q_lvds;

  IBUF u1 (.I(d),.O(d_lvds)) /*synthesis xc_props="IOSTANDARD=LVDS"*/;
  OBUF u2 (.I(q_lvds),.O(q)) /*synthesis xc_props="IOSTANDARD=LVDS"*/;
  OBUF u3 (.I(q_lvds_n),.O(q_n))/*synthesis xc_props=
                                 "IOSTANDARD=LVDS"*/;
```

```
    //synthesis attribute IOSTANDARD u1 "LVDS"
    //synthesis attribute IOSTANDARD u2 "LVDS"
    //synthesis attribute IOSTANDARD u3 "LVDS"

    //exemplar attribute u1 IOSTANDARD LVDS
    //exemplar attribute u2 IOSTANDARD LVDS
    //exemplar attribute u3 IOSTANDARD LVDS

    always @(posedge clk) q_lvds <= d_lvds;
    assign q_lvds_n=~q_lvds;
  endmodule
```

Reference Xilinx® Answer Database in the Xilinx® support website at http://support.xilinx.com for more information.

In LeonardoSpectrum™ and Synplify™, you can instantiate the SelectIO™ components or use the attribute discussed in the "Inputs" section, but make sure that the output and its inversion are declared and configured properly.

## Virtex™-II/II Pro/II Pro X, Spartan-3™ IOBs

Virtex™-II/II Pro/II Pro X or Spartan-3™ offers more SelectIO™ configuration than Virtex™/E and Spartan-II™ as shown in Table 4-3, page 119. IOSTANDARD and synthesis tools' specific attributes can be used to configure the SelectIO™.

Additionally, Virtex™-II/II Pro/II Pro X or Spartan-3™ provides digitally controlled impedance (DCI) I/Os which are useful in improving signal integrity and avoiding the use of external resistors. This option is only available for most of the single ended I/O standards. To access this option you can instantiate the 'DCI' suffixed I/Os from the library such as HSTL_IV_DCI.

For low-voltage differential signaling, additional IBUFDS, OBUFDS, OBUFTDS, and IOBUFDS components are available. These components simplify the task of instantiating the differential signaling standard.

### Differential Signaling in Virtex™-II/II Pro/II Pro X or Spartan-3™

Differential signaling in Virtex™-II/II Pro/II Pro X or Spartan-3™ can be configured using IBUFDS, OBUFDS, and OBUFTDS. The IBUFDS is a two-input one-output buffer. The OBUFDS is a one-input two-output buffer. Refer to the *Libraries Guide* for the component diagram and description.

LVDS_25, LVDS_33, LVDSEXT_33, and LVPECL_33 are valid IOSTANDARD values to attach to differential signaling buffers. If no IOSTANDARD is attached, the default is LVDS_33.

The following is the VHDL and Verilog example of instantiating differential signaling buffers.

- VHDL example

```
--------------------------------------------
-- LVDS_33_IO.VHD Version 1.0             --
-- Example of a behavioral description of --
-- differential signal I/O standard using --
-- LeonardoSpectrum attribute.            --
-- HDL Synthesis Design Guide for FPGAs   --
--------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
--use exemplar.exemplar_1164.all;
entity LVDS_33_IO is
    port(
        CLK_p,CLK_n,DATA_p,DATA_n, Tin_p,Tin_n : in STD_LOGIC;
        datain2_p, datain2_n : in STD_LOGIC;
        ODATA_p, ODATA_n : out STD_LOGIC;
        Q_p, Q_n : out STD_LOGIC
        );
end LVDS_33_IO;
architecture BEHAV of LVDS_33_IO is
  component IBUFDS is
    port (
        I : in STD_LOGIC;
        IB : in STD_LOGIC;
        O : out STD_LOGIC
        );
  end component;
  component OBUFDS is
    port (
        I : in STD_LOGIC;
        O : out STD_LOGIC;
        OB : out STD_LOGIC
        );
  end component;
  component OBUFTDS is
    port (
        I : in STD_LOGIC;
        T : in STD_LOGIC;
        O : out STD_LOGIC;
        OB : out STD_LOGIC
        );
  end component;
  component IBUFGDS is
    port(
        I : in STD_LOGIC;
        IB : in STD_LOGIC;
        O : out STD_LOGIC
        );
  end component;
  component BUFG is
    port(
        I : in STD_LOGIC;
        O :out STD_LOGIC
        );
  end component;
  signal datain2 : std_logic;
  signal odata_out : std_logic;
  signal DATA_int : std_logic;
  signal Q_int : std_logic;
  signal CLK_int : std_logic;
  signal CLK_ibufgout : std_logic;
  signal Tin_int : std_logic;
begin
  UI1 : IBUFDS port map (I => DATA_p, IB => DATA_n, O => DATA_int);
  UI2 : IBUFDS port map (I => datain2_p,IB => datain2_n,O => datain2);
  UI3 : IBUFDS port map (I => Tin_p,IB => Tin_n,O => Tin_int);
  UO1 : OBUFDS port map (I => Q_int,O => Q_p,OB => Q_n);
  UO2 : OBUFTDS port map (
       I => odata_out,
       T => Tin_int,
       O => odata_p,
       OB => odata_n
       );
  UIBUFG : IBUFGDS port map (I => CLK_p,IB => CLK_n,O => CLK_ibufgout);
  UBUFG  : BUFG port map (I => CLK_ibufgout,O => CLK_int);
  My_D_Reg: process (CLK_int, DATA_int)
    begin
     if (CLK_int’event and CLK_int=’1’) then
        Q_int <= DATA_int;
     end if;
  end process;       -- End My_D_Reg
  odata_out <= DATA_int and datain2;
end BEHAV;
```

- Verilog example

```
//-----------------------------------------
// LVDS_33_IO.v Version 1.0              --
// Example of a behavioral description of --
// differential signal I/O standard      --
// HDL Synthesis Design Guide for FPGAs   --
//-----------------------------------------
module LVDS_33_IO (CLK_p, CLK_n, DATA_p, DATA_n, DATAIN2_p, DATAIN2_n,
                  Tin_p, Tin_n, ODATA_p, ODATA_n, Q_p, Q_n);
input CLK_p, CLK_n, DATA_p, DATA_n, DATAIN2_p, DATAIN2_n, Tin_p, Tin_n;
output ODATA_p, ODATA_n;
output Q_p, Q_n;
wire datain2;
wire odata_in;
wire odata_out;
wire DATA_int;
reg Q_int;
wire CLK_int;
wire CLK_ibufgout;
wire Tin_int;
IBUFDS UI1 (
       .I (DATA_p),
       .IB(DATA_n),
       .O (DATA_int)
       );
IBUFDS UI2 (
       .I (Tin_p),
       .IB(Tin_n),
       .O (Tin_int)
       );
IBUFDS UI3 ( .I(DATAIN2_p), .IB(DATAIN2_n), .O(datain2));
OBUFDS UO1 ( .I(Q_int), .O(Q_p), .OB(Q_n));
OBUFTDS UO2 ( .I(odata_out), .T(Tin_int), .O(ODATA_p), .OB(ODATA_n));
IBUFGDS UIBUFG ( .I(CLK_p), .IB(CLK_n), .O(CLK_ibufgout));
BUFG UBUFG ( .I(CLK_ibufgout), .O(CLK_int));
always @ (posedge CLK_int)
  begin
    Q_int <= DATA_int;
  end
assign odata_out = DATA_int && datain2;
endmodule
```

# Encoding State Machines

The traditional methods used to generate state machine logic result in highly-encoded states. State machines with highly-encoded state variables typically have a minimum number of flip-flops and wide combinatorial functions. These characteristics are acceptable for PAL and gate array architectures. However, because FPGAs have many flip-flops and narrow function generators, highly-encoded state variables can result in inefficient implementation in terms of speed and density.

One-hot encoding allows you to create state machine implementations that are more efficient for FPGA architectures. You can create state machines with one flip-flop per state and decreased width of combinatorial logic. One-hot encoding is usually the preferred method for large FPGA-based state machine implementation. For small state machines (fewer than 8 states), binary encoding may be more efficient. To improve design performance, you can divide large (greater than 32 states) state machines into several small state machines and use the appropriate encoding style for each.

Three design examples are provided in this section to illustrate the three coding methods (binary, enumerated type, and one-hot) you can use to create state machines. All three examples contain the same Case statement. To conserve space, the complete Case statement is only included in the binary encoded state machine example; refer to this example when reviewing the enumerated type and one-hot examples.

Some synthesis tools allow you to add an attribute, such as TYPE_ENCODING_STYLE, to your VHDL code to set the encoding style. This is a synthesis vendor attribute (not a Xilinx® attribute). Refer to your synthesis tool documentation for information on attribute-driven state machine synthesis.

## Using Binary Encoding

The state machine bubble diagram in the following figure shows the operation of a seven-state machine that reacts to inputs A through E as well as previous-state conditions. The binary encoded method of coding this state machine is shown in the VHDL and Verilog examples that follow. These design examples show you how to take a design that has been previously encoded (for example, binary encoded) and synthesize it to the appropriate decoding logic and registers. These designs use three flip-flops to implement seven states.

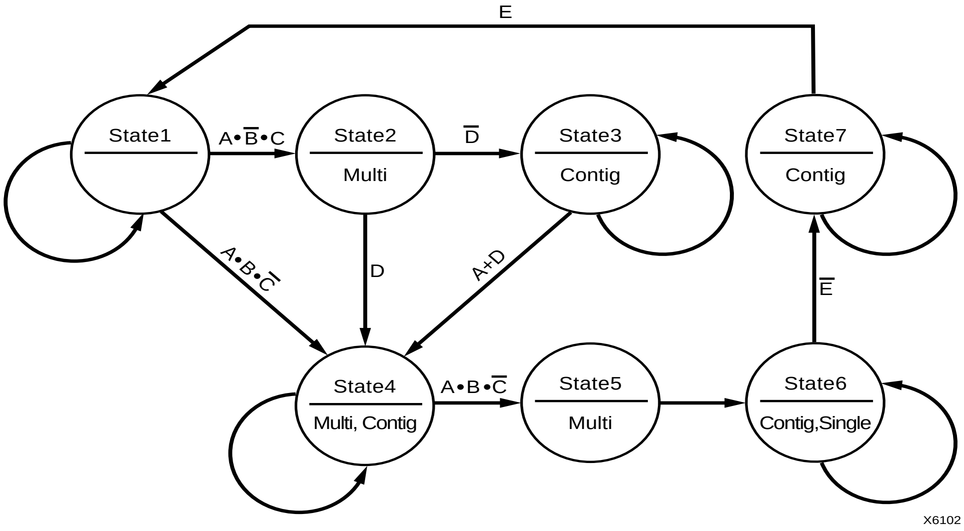


*Figure 4-4:* **State Machine Bubble Diagram**

## Binary Encoded State Machine VHDL Example

The following is a binary encoded state machine VHDL example.

```
-------------------------------------------------
-- BINARY.VHD Version 1.0                      --
-- Example of a binary encoded state machine   --
-------------------------------------------------
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity binary is
  port (
        CLOCK, RESET : in STD_LOGIC;
        A, B, C, D, E : in BOOLEAN;
        SINGLE, MULTI, CONTIG : out STD_LOGIC
        );
end binary;

architecture BEHV of binary is

  type STATE_TYPE is (S1, S2, S3, S4, S5, S6, S7);
  attribute ENUM_ENCODING: STRING;
  attribute ENUM_ENCODING of STATE_TYPE:type is
        "001 010 011 100 101 110 111";
  signal CS, NS: STATE_TYPE;

  begin
    SYNC_PROC: process (CLOCK, RESET)
    begin
      if (RESET='1') then
          CS <= S1;
      elsif (CLOCK'event and CLOCK = '1') then
          CS <= NS;
      end if;
    end process; --End REG_PROC

    COMB_PROC: process (CS, A, B, C, D, E)
    begin
      case CS is
        when S1 =>
            MULTI <= '0';
            CONTIG <= '0';
            SINGLE <= '0';
            if (A and not B and C) then
              NS <= S2;
            elsif (A and B and not C) then
              NS <= S4;
            else
              NS <= S1;
            end if;
        when S2 =>
            MULTI <= ’1’;
            CONTIG <= ’0’;
            SINGLE <= ’0’;
            if (not D) then
                NS <= S3;
            else
                NS <= S4;
            end if;

        when S3 =>
            MULTI <= ’0’;
            CONTIG <= ’1’;
            SINGLE <= ’0’;
            if (A or D) then
                NS <= S4;
            else
                NS <= S3;
            end if;

        when S4 =>
            MULTI <= ’1’;
            CONTIG <= ’1’;
            SINGLE <= ’0’;
            if (A and B and not C) then
                NS <= S5;
            else
                NS <= S4;
            end if;

        when S5 =>
            MULTI <= ’1’;
            CONTIG <= ’0’;
            SINGLE <= ’0’;
            NS <= S6;

        when S6 =>
            MULTI <= ’0’;
            CONTIG <= ’1’;
            SINGLE <= ’1’;
            if (not E) then
                NS <= S7;
            else
                NS <= S6;
            end if;

        when S7 =>
            MULTI <= ’0’;
            CONTIG <= ’1’;
            SINGLE <= ’0’;
            if (E) then
                NS <= S1;
            else
                NS <= S7;
            end if;
    end case;
  end process; -- End COMB_PROC
end BEHV;
```

## Binary Encoded State Machine Verilog Example

```
/////////////////////////////////////////////////
// BINARY.V Version 1.0                        //
// Example of a binary encoded state machine   //
/////////////////////////////////////////////////
module binary (CLOCK, RESET, A, B, C, D, E, SINGLE, MULTI, CONTIG);
input  CLOCK, RESET;
input  A, B, C, D, E;
output SINGLE, MULTI, CONTIG;
reg    SINGLE, MULTI, CONTIG;
// Declare the symbolic names for states
parameter [2:0]
   S1 = 3’b001,
   S2 = 3’b010,
   S3 = 3’b011,
   S4 = 3’b100,
   S5 = 3’b101,
   S6 = 3’b110,
   S7 = 3’b111;

// Declare current state and next state variables
reg [2:0] CS;
reg [2:0] NS;

// state_vector CS
  always @ (posedge CLOCK or posedge RESET)
    begin
      if (RESET == 1’b1)
        CS <= S1;
      else
        CS <= NS;
    end
  always @ (CS or A or B or C or D or D or E)
    begin
      case (CS)
        S1 :
          begin
            MULTI = 1’b0;
            CONTIG = 1’b0;
            SINGLE = 1’b0;
            if (A && ~B && C)
                NS = S2;
            else if (A && B && ~C)
                NS = S4;
            else
              NS = S1;
          end

        S2 :
          begin
            MULTI = 1’b1;
            CONTIG = 1’b0;
            SINGLE = 1’b0;
            if (!D)
                NS = S3;
            else
                NS = S4;
          end
        S3 :
          begin
            MULTI = 1’b0;
            CONTIG = 1’b1;
            SINGLE = 1’b0;
            if (A || D)
                NS = S4;
            else
              NS = S3;
          end

        S4 :
          begin
            MULTI = 1’b1;
            CONTIG = 1’b1;
            SINGLE = 1’b0;
            if (A && B && ~C)
                NS = S5;
            else
                NS = S4;
          end

        S5 :
          begin
            MULTI = 1’b1;
            CONTIG = 1’b0;
            SINGLE = 1’b0;
            NS = S6;
          end

        S6 :
          begin
            MULTI = 1’b0;
            CONTIG = 1’b1;
            SINGLE = 1’b1;
            if (!E)
                NS = S7;
            else
                NS = S6;
          end

        S7 :
          begin
            MULTI = 1’b0;
            CONTIG = 1’b1;
            SINGLE = 1’b0;
            if (E)
                NS = S1;
            else
                NS = S7;
          end
      endcase
    end
endmodule
```

## Using Enumerated Type Encoding

The recommended encoding style for state machines depends on which synthesis tool you are using. Some synthesis tools encode better than others depending on the device architecture and the size of the decode logic. You can explicitly declare state vectors or you can allow your synthesis tool to determine the vectors. Xilinx® recommends that you use enumerated type encoding to specify the states and use the Finite State Machine (FSM) extraction commands to extract and encode the state machine as well as to perform state minimization and optimization algorithms. The enumerated type method of encoding the seven-state machine is shown in the following VHDL and Verilog examples. The encoding style is not defined in the code, but can be specified later with the FSM extraction commands. Alternatively, you can allow your compiler to select the encoding style that results in the lowest gate count when the design is synthesized. Some synthesis tools automatically find finite state machines and compile without the need for specification.

***Note:*** Refer to the previous VHDL and Verilog Binary Encoded State Machine examples for the complete Case statement portion of the code.

### Enumerated Type Encoded State Machine VHDL Example

```
Library IEEE;
use IEEE.std_logic_1164.all;
entity enum is
  port (
        CLOCK, RESET : in STD_LOGIC;
        A, B, C, D, E : in BOOLEAN;
        SINGLE, MULTI, CONTIG : out STD_LOGIC
        );
end enum;

architecture BEHV of enum is

  type STATE_TYPE is (S1, S2, S3, S4, S5, S6, S7);
  signal CS, NS: STATE_TYPE;

  begin
    SYNC_PROC: process (CLOCK, RESET)
    begin
      if (RESET='1') then
          CS <= S1;
      elsif (CLOCK'event and CLOCK = '1') then
          CS <= NS;
      end if;
    end process; --End SYNC_PROC
    COMB_PROC: process (CS, A, B, C, D, E)
    begin
      case CS is
          when S1 =>
              MULTI <= '0';
              CONTIG <= '0';
              SINGLE <= '0';
.
.
.
```

## Enumerated Type Encoded State Machine Verilog Example

```
/////////////////////////////////////////////////////
// ENUM.V Version 1.0                              //
// Example of an enumerated encoded state machine  //
/////////////////////////////////////////////////////

module enum (CLOCK, RESET, A, B, C, D, E, SINGLE, MULTI, CONTIG);

input CLOCK, RESET;
input A, B, C, D, E;
output SINGLE, MULTI, CONTIG;

reg    SINGLE, MULTI, CONTIG;

// Declare the symbolic names for states
parameter [2:0]
    S1 = 3’b000,
    S2 = 3’b001,
    S3 = 3’b010,
    S4 = 3’b011,
    S5 = 3’b100,
    S6 = 3’b101,
    S7 = 3’b110;

// Declare current state and next state variables
reg [2:0] CS;
reg [2:0] NS;

// state_vector CS
always @ (posedge CLOCK or posedge RESET)
  begin
    if (RESET == 1’b1)
        CS <= S1;
    else
        CS <= NS;
  end

always @ (CS or A or B or C or D or D or E)
  begin
    case (CS)
      S1 :
      begin
        MULTI = 1’b0;
        CONTIG = 1’b0;
        SINGLE = 1’b0;
        if (A && ~B && C)
            NS = S2;
        else if (A && B && ~C)
            NS = S4;
        else
            NS = S1;
      end
.
.
.
```

## Using One-Hot Encoding

One-hot encoding allows you to create state machine implementations that are more efficient for FPGA architectures. One-hot encoding is usually the preferred method for large FPGA-based state machine implementation.

The following examples show a one-hot encoded state machine. Use this method to control the state vector specification or when you want to specify the names of the state registers. These examples use one flip-flop for each of the seven states. If you are using FPGA Compiler II™, use enumerated type, and avoid using the "when others" construct in the VHDL Case statement. This construct can result in a very large state machine.

***Note:*** Refer to the previous VHDL and Verilog Binary Encoded State Machine examples for the complete Case statement portion of the code.

### One-hot Encoded State Machine VHDL Example

```
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity one_hot is
  port (
        CLOCK, RESET : in STD_LOGIC;
        A, B, C, D, E : in BOOLEAN;
        SINGLE, MULTI, CONTIG : out STD_LOGIC
        );
end one_hot;

architecture BEHV of one_hot is
  type STATE_TYPE is (S1, S2, S3, S4, S5, S6, S7);
  attribute ENUM_ENCODING: STRING;
  attribute ENUM_ENCODING of STATE_TYPE: type is
    "0000001 0000010 0000100 0001000 0010000 0100000 1000000 ";
  signal CS, NS: STATE_TYPE;

  begin
    SYNC_PROC: process (CLOCK, RESET)
    begin
      if (RESET='1') then
          CS <= S1;
    elsif (CLOCK'event and CLOCK = '1') then
          CS <= NS;
    end if;
    end process;     --End SYNC_PROC
```

```
   COMB_PROC: process (CS, A, B, C, D, E)
     begin
       case CS is
         when S1 =>
            MULTI <= ’0’;
            CONTIG <= ’0’;
            SINGLE <= ’0’;
            if (A and not B and C) then
               NS <= S2;
            elsif (A and B and not C) then
               NS <= S4;
            else
               NS <= S1;
            end if;
          .
          .
          .
```

## One-hot Encoded State Machine Verilog Example

```
///////////////////////////////////////////////////
// ONE_HOT.V Version 1.0                         //
// Example of a one-hot encoded state machine    //
// Xilinx HDL Synthesis Design Guide for FPGAs   //
///////////////////////////////////////////////////

module one_hot (CLOCK, RESET, A, B, C, D, E, SINGLE, MULTI, CONTIG);

input  CLOCK, RESET;
input  A, B, C, D, E;
output SINGLE, MULTI, CONTIG;

reg    SINGLE, MULTI, CONTIG;

// Declare the symbolic names for states
parameter [6:0]
   S1 = 7’b0000001,
   S2 = 7’b0000010,
   S3 = 7’b0000100,
   S4 = 7’b0001000,
   S5 = 7’b0010000,
   S6 = 7’b0100000,
   S7 = 7’b1000000;

// Declare current state and next state variables
reg [2:0] CS;
reg [2:0] NS;

// state_vector CS

  always @ (posedge CLOCK or posedge RESET)
   begin
     if (RESET == 1’b1)
       CS <= S1;
     else
       CS <= NS;
   end
```

```
always @ (CS or A or B or C or D or D or E)
  begin
    case (CS)
      S1 :
      begin
        MULTI = 1’b0;
        CONTIG = 1’b0;
        SINGLE = 1’b0;
        if (A && ~B && C)
            NS = S2;
        else if (A && B && ~C)
            NS = S4;
        else
            NS = S1;
      end
      .
      .
      .
```

## Accelerating FPGA Macros with One-Hot Approach

Most synthesis tools provide a setting for finite state machine (FSM) encoding. This setting prompts synthesis tools to automatically extract state machines in your design and perform special optimizations to achieve better performance. The default option for FSM encoding is “One-Hot” for most synthesis tools. However, this setting can be changed to other encoding such as binary, gray, sequential, etc.

In FPGA Compiler II™, FSM encoding is set to “One-Hot” by default. To change this setting, select **Synthesis**→ **Options**→ **Project** Tab. Available options are: One-Hot, Binary, and Zero One-Hot.

In LeonardoSpectrum™, FSM encoding is set to “Auto” by default, which differs depending on the Bit Width of your state machine. To change this setting to a specific value, select the Input tab. Available options are: Binary, One-Hot, Random, Gray, and Auto.

In Synplify™, the Symbolic FSM Complier option can be accessed from the main GUI. When set, the FSM Compiler extracts the state machines as symbolic graphs, and then optimizes them by re-encoding the state representations and generating a better logic optimization starting point for the state machines. This usually results in one-hot encoding. However, you may override the default on a register by register basis with the SYN_ENCODING directive/attribute. Available options are: One-Hot, Gray, Sequential, and Safe.

In XST, FSM encoding is set to Auto by default. Available options are: Auto, One-Hot, Compact, Gray, Johnson, Sequential, and User.

***Note:*** XST only recognizes enumerated encoding if the encoding option is set to User.

## Summary of Encoding Styles

In the three previous examples, the state machine’s possible states are defined by an enumeration type. Use the following syntax to define an enumeration type.

**`type`** *type_name* **`is`** (*enumeration_literal* {, *enumeration_literal*} );

After you have defined an enumeration type, declare the signal representing the states as the enumeration type as follows.

```
type STATE_TYPE is (S1, S2, S3, S4, S5, S6, S7);
signal CS, NS: STATE_TYPE;
```

The state machine described in the three previous examples has seven states. The possible values of the signals CS (Current_State) and NS (Next_State) are S1, S2, ... , S6, S7.

To select an encoding style for a state machine, specify the state vectors. Alternatively, you can specify the encoding style when the state machine is compiled. Xilinx® recommends that you specify an encoding style. If you do not specify a style, your compiler selects a style that minimizes the gate count. For the state machine shown in the three previous examples, the compiler selected the binary encoded style: S1="000", S2="001", S3="010", S4="011", S5="100", S6="101", and S7="110".

You can use the FSM extraction tool to change the encoding style of a state machine. For example, use this tool to convert a binary-encoded state machine to a one-hot encoded state machine.

***Note:*** Refer to your synthesis tool documentation for instructions on how to extract the state machine and change the encoding style.

## Initializing the State Machine

When creating a state machine, especially when you use one-hot encoding, add the following lines of code to your design to ensure that the FPGA is initialized to a Set state.

- VHDL example

```
SYNC_PROC: process (CLOCK, RESET)
begin
  if (RESET='1') then
      CS <= s1;
```

- Verilog example

```
always @ (posedge CLOCK or posedge RESET)
  begin
    if (RESET == 1'b 1)
      CS <= S1;
```

Alternatively, you can assign an INIT=S attribute to the initial state register to specify the initial state. Refer to your synthesis tool documentation for information on assigning this attribute.

In the Binary Encode State Machine example, the RESET signal forces the S1 flip-flop to be preset (initialized to 1) while the other flip-flops are cleared (initialized to 0).

# Implementing Operators and Generate Modules

Xilinx® FPGAs feature carry logic elements that can be used for optimal implementation of operators and generate modules. Synthesis tools infer the carry logic automatically when a specific coding style or operator is used.

## Adder and Subtractor

Synthesis tools infer carry logic in Virtex™/E/II/II Pro/II Pro X and Spartan™-II/3 devices when an adder and Subtractor is described (+ or - operator).

## Multiplier

Synthesis tools utilize carry logic by inferring XORCY, MUXCY, and MULT_AND for Virtex™, Virtex-E™ and Spartan-II™ when a multiplier is described.

When a Virtex™-II / II Pro part is targeted an embedded 18x18 two's complement multiplier primitive called a MULT18X18 is inferred by the synthesis tools. For synchronous multiplications, LeonardoSpectrum™, Synplify™ and XST infer a MULT18X18S primitive.

LeonardoSpectrum™ also features a pipeline multiplier that involves putting levels of registers in the logic to introduce parallelism and, as a result, improve speed. A certain construct in the input RTL source code description is required to allow the pipelined multiplier feature to take effect. This construct infers XORCY, MUXCY, and MULT_AND primitives for Virtex™, Virtex-E™, Spartan-II™, Spartan-3™,Virtex-II™, Virtex-II Pro™ and Virtex™-Pro X. The following example shows this construct.

- VHDL example

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity multiply is
generic (size : integer := 16; level : integer := 4);
  port (
        clk : in std_logic;
        Ain : in std_logic_vector (size-1 downto 0);
        Bin : in std_logic_vector (size-1 downto 0);
        Qout : out std_logic_vector (2*size-1 downto 0)
        );
end multiply;
architecture RTL of multiply is
  type levels_of_registers is array (level-1 downto 0)
    of unsigned (2*size-1 downto 0);
  signal reg_bank :levels_of_registers;
  signal a, b : unsigned (size-1 downto 0);
  begin
    Qout <= std_logic_vector (reg_bank (level-1));
    process
    begin
      wait until clk'event and clk = '1';
        a <= unsigned(Ain);
        b <= unsigned(Bin);
        reg_bank (0) <= a * b;
        for i in 1 to level-1 loop
            reg_bank (i) <= reg_bank (i-1);
        end loop;
    end process;
  end architecture RTL;
```

The following is a synchronous multiplier VHDL example coded for Synplify™ and XST:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity xcv2_mult18x18s is
  Port (
        a : in std_logic_vector(7 downto 0);
        b : in std_logic_vector(7 downto 0);
        clk : in std_logic;
        prod : out std_logic_vector(15 downto 0)
        );
end xcv2_mult18x18s;

architecture arch_ xcv2_mult18x18s of xcv2_mult18x18s is

  begin
  process(clk) is begin
    if clk’event and clk = ’1’ then
      prod <= a*b;
    end if;
  end process;
  end arch_ xcv2_mult18x18s;
```

The following is a synchronous multiplier VHDL example coded for LeonardoSpectrum™:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity xcv2_mult18x18s is
  port(
        clk: in std_logic;
        a: in std_logic_vector(7 downto 0);
        b: in std_logic_vector(7 downto 0);
        prod: out std_logic_vector(15 downto 0));
end  xcv2_mult18x18s;

architecture arch_ xcv2_mult18x18s of xcv2_mult18x18 is
  signal reg_prod : std_logic_vector(15 downto 0);
  begin
    process(clk)
      begin
        if(rising_edge(clk))then
          reg_prod <= a * b;
          prod <= reg_prod;
        end if;
    end process;
  end arch_ xcv2_mult18x18s;
```

- Verilog example

```
module multiply (clk, ain, bin, q);
  parameter size = 16;
  parameter level = 4;
  input clk;
  input [size-1:0] ain, bin;
  output [2*size-1:0] q;
  reg [size-1:0] a, b;
  reg [2*size-1:0] reg_bank [level-1:0];
  integer i;
  always @(posedge clk)
    begin
      a <= ain;
      b <= bin;
    end
  always @(posedge clk)
    reg_bank[0] <= a * b;
  always @(posedge clk)
    for (i = 1;i < level; i=i+1)
      reg_bank[i] <= reg_bank[i-1];
    assign q = reg_bank[level-1];
endmodule // multiply
```

The following is a synchronous multiplier Verilog example coded for Synplify™ and XST:

```
module mult_sync(a,b,clk,prod);
  input [7:0] a;
  input [7:0] b;
  input clk;
  output [15:0] prod;
  reg [15:0] prod;

  always @(posedge clk)
    prod <= a*b;
endmodule
```

The following is a synchronous multiplier Verilog example coded for LeonardoSpectrum™:

```
module xcv2_mult18x18s (a,b,clk,prod);
  input [7:0] a;
  input [7:0] b;
  input clk;
  output [15:0] prod;
  reg [15:0] reg_prod, prod;

  always @(posedge clk) begin
    reg_prod <= a*b;
    prod <= reg_prod;
endmodule
```

## Counters

When describing a counter in HDL, the arithmetic operator ’+’ infers the carry chain. The synthesis tools then infers the MUXCY element for the counter.

```
count <= count + 1; -- This infers MUXCY
```

This implementation provides a very effective solution, especially for all purpose counters.

Following is an example of a loadable binary counter:

- VHDL example

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counter is
  port (
        d : in std_logic_vector (7 downto 0);
        ld, ce, clk, rst : in std_logic;
        q : out std_logic_vector (7 downto 0)
        );
end counter;

architecture behave of counter is
  signal count : std_logic_vector (7 downto 0);
begin
  process (clk, rst) begin
    if rst = ’1’ then
      count <= (others => ’0’);
    elsif rising_edge(clk) then
      if ld = ’1’ then
        count <= d;
      elsif ce = ’1’ then
        count <= count + ’1’;
      end if;
    end if;
  end process;
  q <= count;
end behave;
```

- Verilog example

```
module counter(d, ld, ce, clk, rst, q);
  input [7:0] d;
  input ld, ce, clk, rst;
  output [7:0] q;
  reg [7:0] count;
  always @(posedge clk or posedge rst)
    begin
        if (rst)
          count <= 0;
        else if (ld)
          count <= d;
        else if (ce)
          count <= count + 1;
    end
  assign q = count;
endmodule
```

For applications that require faster counters, LFSR can implement high performance and area efficient counters. LFSR requires very minimal logic (only an XOR or XNOR feedback).

For smaller counters it is also effective to use the Johnson encoded counters. This type of counter does not use the carry chain but provides a fast performance.

The following is an example of a sequence for a 3 bit johnson counter.

000

001

011

111

110

100

- VHDL example

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity johnson is
  generic (size : integer := 3);
  port (
        clk : in std_logic;
        reset : in std_logic;
        qout : out std_logic_vector(size-1 downto 0)
        );
end johnson;
architecture RTL of johnson is
  signal q : std_logic_vector(size-1 downto 0);
  begin   -- RTL
    process(clk, reset)
    begin
      if reset = ’1’ then
          q <= (others => ’0’);
      elsif clk’event and clk = ’1’ then
          for i in 1 to size - 1 loop
              q(i) <= q(i-1);
          end loop;   -- i
          q(0) <= not q(size-1);
      end if;
    end process;
    qout <= q;
end RTL;
```

- Verilog example

```
module johnson (clk, reset, q);
parameter size = 4;
  input clk, reset;
  output [size-1:0] q;
  reg [size-1:0] q;
  integer i;
  always @(posedge clk or posedge reset)
    if (reset)
      q <= 0;
    else
      begin
        for (i=1;i<size;i=i+1)
            q[i] <= q[i-1];
            q[0] <= ~q[size-1];
      end
endmodule // johnson
```

## Comparator

Magnitude comparators ’>’ or ’<’ infer carry chain logic and result in fast implementations in Xilinx® devices. Equality comparator '==' is implemented using LUTs.

- VHDL example

```
-- Unsigned 8-bit greater or equal comparator.
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity compar is
   port(A,B : in std_logic_vector(7 downto 0);
      cmp : out std_logic);
end compar;

architecture archi of compar is
begin
   cmp <= ’1’ when A >= B else ’0’;
end archi;
```

- Verilog example

```
// Unsigned 8-bit greater or equal comparator.
module compar(A, B, cmp);
input [7:0] A;
input [7:0] B;
output cmp;
assign cmp = A >= B ? 1’b1 : 1’b0;
endmodule
```

## Encoder and Decoders

Synthesis tools might infer MUXF5 and MUXF6 for encoder and decoder in Virtex™/E/II/II Pro/II Pro X and Spartan™-II/3 devices. Virtex™-II/II Pro/II Pro X or Spartan-3™ devices feature additional components, MUXF7 and MUXF8 to use with the encoder and decoder.

LeonardoSpectrum™ infers MUXCY when an if-then-else priority encoder is described in the code. This results in a faster encoder.

### LeonardoSpectrum™ Priority Encoding HDL Example

- VHDL example

```
library IEEE;
use IEEE.std_logic_1164.all;
entity prior is
  generic (size: integer := 8);
    port(
        clk: in std_logic;
        cond1 : in std_logic_vector(size downto 1);
        cond2 : in std_logic_vector(size downto 1);
        data  : in std_logic_vector(size downto 1);
        q     : out std_logic);
end prior;

architecture RTL of prior is
  signal  data_ff, cond1_ff, cond2_ff: std_logic_vector(size downto 1);
  begin
    process(clk)
    begin
      if clk’event and clk= ’1’ then
        data_ff <= data;
        cond1_ff <= cond1;
        cond2_ff <= cond2;
      end if;
    end process;
    process(clk)
    begin
      if (clk’event and clk = ’1’) then
        if (cond1_ff(1) = ’1’ and cond2_ff(1) = ’1’) then
          q <= data_ff(1);
        elsif (cond1_ff(2) = ’1’ and cond2_ff(2) = ’1’) then
          q <= data_ff(2);
        elsif (cond1_ff(3) = ’1’ and cond2_ff(3) = ’1’) then
          q <= data_ff(3);
        elsif (cond1_ff(4) = ’1’ and cond2_ff(4) = ’1’) then
          q <= data_ff(4);
        elsif (cond1_ff(5)= ’1’ and cond2_ff(5) = ’1’ ) then
          q <= data_ff(5);
        elsif (cond1_ff(6) = ’1’ and cond2_ff(6) = ’1’) then
          q <= data_ff(6);
        elsif (cond1_ff(7) = ’1’ and cond2_ff(7) = ’1’) then
          q <= data_ff(7);
        elsif (cond1_ff(8) = ’1’ and cond2_ff(8) = ’1’) then
          q <= data_ff(8);
        else
          q <= ’0’;
        end if;
      end if;
  end process;
end RTL;
```

- Verilog example

```
module prior(clk, cond1, cond2, data, q);
  parameter size = 8;
  input clk;
  input [1:size] data, cond1, cond2;
  output q;
  reg [1:size] data_ff, cond1_ff, cond2_ff;
  reg q;
  always @(posedge clk)
    begin
      data_ff = data;
      cond1_ff = cond1;
      cond2_ff = cond2;
    end
  always @(posedge clk)
    if (cond1_ff[1] && cond2_ff[1])
        q <= data_ff[1];
    else if (cond1_ff[2] && cond2_ff[2])
        q <= data_ff[2];
    else if (cond1_ff[3] && cond2_ff[3])
        q <= data_ff[3];
    else if (cond1_ff[4] && cond2_ff[4])
        q <= data_ff[4];
    else if (cond1_ff[5] && cond2_ff[5])
        q <= data_ff[5];
    else if (cond1_ff[6] && cond2_ff[6])
        q <= data_ff[6];
    else if (cond1_ff[7] && cond2_ff[7])
        q <= data_ff[7];
    else if (cond1_ff[8] && cond2_ff[8])
        q <= data_ff[8];
    else q <= 1’b0;
endmodule // prior
```

# Implementing Memory

Virtex™/E and Spartan-II™ FPGAs provide distributed on-chip RAM or ROM memory capabilities. CLB function generators can be configured as:

- ROM (ROM16X1, ROM32X1)
- edge-triggered, single-port (RAM16X1S, RAM32X1S) RAM
- dual-port (RAM16x1D) RAM.

Level sensitive RAMs are not available for the Virtex™/E and Spartan-II™ families.

Virtex™-II/II Pro/II Pro X or Spartan-3™ CLB function generators are much larger and can be configured as larger ROM and edge-triggered, single port and dual port RAM. Available ROM primitive components in Virtex™-II/II Pro/II Pro X or Spartan-3™ are:

- ROM16X1
- ROM32X1.

Available single port RAM primitives components in Virtex™-II / II Pro / II Pro X or Spartan-3™ are:

RAM16X1S
RAM16X2S
RAM16X4S
RAM16X8S
RAM32X1S
RAM32X2S
RAM32X4S
RAM32X8S
RAM64X1S
RAM64X2S
RAM128X1S

Available dual port RAM primitive components in Virtex™-II / II Pro / II Pro X or Spartan-3™ are as follows:

- RAM16X1D
- RAM32X1D
- RAM64X1D

In addition to distributed RAM and ROM capabilities, Virtex™ / E / II and Spartan-II™ FPGAs provide edge-triggered Block SelectRAM™ in large blocks. Virtex™ / E and Spartan-II™ devices provide 4096(4k) bits: RAMB4_Sn and RAMB4_Sm_Sn. Virtex™-II / II Pro / II Pro X and Spartan-3™ devices provide larger Block SelectRAM™ in 16384 (16k) bit size: RAMB16_Sn and RAMB16_Sm_Sn, where Sm and Sn are configurable port widths. See the *Libraries Guide* for more information on these components.

The edge-triggered capability simplifies system timing and provides better performance for RAM-based designs. This RAM can be used for status registers, index registers, counter storage, constant coefficient multipliers, distributed shift registers, LIFO stacks, latching or any data storage operation. The dual-port RAM simplifies FIFO designs.

## Implementing Block RAM

Virtex™, Virtex-E™, Virtex-II™, Virtex-II Pro™, Virtex-II Pro X™, Spartan-II™ and Spartan-3™ FPGAs incorporate several large Block SelectRAM™ memories. These complement the distributed SelectRAM™ that provide shallow RAM structures implemented in CLBs. The Block SelectRAM™ is a True Dual-Port RAM which allows for large, discrete blocks of memory.

Block SelectRAM™ memory blocks are organized in columns. All Virtex™ and Spartan-II™ devices contain two such columns, one along each vertical edge. In Virtex-E™, the Block SelectRAM™ column is inserted every 12 CLB columns. In Virtex-EM™ (Virtex-E™ with extended memory), the Block SelectRAM™ column is inserted every 4 CLB columns. In Virtex-II™, Virtex-II Pro™, Virtex-II Pro X™ or Spartan-3™, there are at least four Block SelectRAM™ columns and a column is inserted every 12 CLB columns in larger devices.

### Instantiating Block SelectRAM™

The following coding examples provide VHDL and Verilog coding styles for FPGA Compiler II™, LeonardoSpectrum™, Synplify™, and XST.

#### Instantiating Block SelectRAM™ VHDL Example

- FPGA Compiler II™, LeonardoSpectrum™, and XST

  With FPGA Compiler II™, LeonardoSpectrum™, and XST you can instantiate a RAMB* cell as a blackbox. The INIT_** attribute can be passed as a string in the HDL file as well as the script file. The following VHDL code example shows how to pass INIT in the VHDL file.

- XST

  XST allows you to pass the INIT values through VHDL generics.

- LeonardoSpectrum™

  With LeonardoSpectrum™, in addition to passing the INIT string in HDL, you can pass an INIT string in a LeonardoSpectrum™ command script. The following code sample illustrates this method.

```
set_attribute -instance "inst_ramb4_s4" -name
INIT_00 -type string -value
"1F1E1D1C1B1A191817161514131211100F0E0D0C0B0A09
80706050403020100"

library IEEE;
use IEEE.std_logic_1164.all;

entity spblkrams is
  port(
       CLK  : in std_logic;
       EN   : in std_logic;
       RST  : in std_logic;
       WE   : in std_logic;
       ADDR : in std_logic_vector(11 downto 0);
       DI   : in std_logic_vector(15 downto 0);
       DORAMB4_S4 : out std_logic_vector(3 downto 0);
       DORAMB4_S8 : out std_logic_vector(7 downto 0)
       );
end;
architecture struct of spblkrams is
component RAMB4_S4
  port (
       DI : in STD_LOGIC_VECTOR (3 downto 0);
       EN : in STD_ULOGIC;
       WE : in STD_ULOGIC;
       RST : in STD_ULOGIC;
       CLK : in STD_ULOGIC;
       ADDR : in STD_LOGIC_VECTOR (9 downto 0);
       DO : out STD_LOGIC_VECTOR (3 downto 0)
       );
end component;
component RAMB4_S8
  port (
       DI : in STD_LOGIC_VECTOR (7 downto 0);
       EN : in STD_ULOGIC;
       WE : in STD_ULOGIC;
       RST : in STD_ULOGIC;
       CLK : in STD_ULOGIC;
       ADDR : in STD_LOGIC_VECTOR (8 downto 0);
       DO : out STD_LOGIC_VECTOR (7 downto 0)
       );
end component;
attribute INIT_00: string;
attribute INIT_00 of INST_RAMB4_S4: label is
  "1F1E1D1C1B1A191817161514131211100F0E0D0C0B0A0980706050403020100";
attribute INIT_00 of INST_RAMB4_S8: label is
  "1F1E1D1C1B1A191817161514131211100F0E0D0C0B0A0980706050403020100";
```

```
begin
   INST_RAMB4_S4 : RAMB4_S4 port map (
       DI => DI(3 downto 0),
       EN => EN,
       WE => WE,
       RST => RST,
       CLK => CLK,
       ADDR => ADDR(9 downto 0),
       DO => DORAMB4_S4
       );
   INST_RAMB4_S8 : RAMB4_S8 port map (
       DI => DI(7 downto 0),
       EN => EN,
       WE => WE,
       RST => RST,
       CLK => CLK,
       ADDR => ADDR(8 downto 0),
       DO => DORAMB4_S8
       );
end struct;
```

- XST

```
library IEEE;
use IEEE.std_logic_1164.all;
entity spblkrams is
  port(
       CLK : in std_logic;
       EN : in std_logic;
       RST : in std_logic;
       WE : in std_logic;
       ADDR : in std_logic_vector(11 downto 0);
       DI : in std_logic_vector(15 downto 0);
       DORAMB4_S4 : out std_logic_vector(3 downto 0);
       DORAMB4_S8 : out std_logic_vector(7 downto 0)
       );
end;

architecture struct of spblkrams is
  component RAMB4_S4
   generic( INIT_00 : string :=
  "0000000000000000000000000000000000000000000000000000000000000000");
   port (
       DI : in STD_LOGIC_VECTOR (3 downto 0);
       EN : in STD_ULOGIC;
       WE : in STD_ULOGIC;
       RST : in STD_ULOGIC;
       CLK : in STD_ULOGIC;
       ADDR : in STD_LOGIC_VECTOR (9 downto 0);
       DO : out STD_LOGIC_VECTOR (3 downto 0)
       );
  end component;
  component RAMB4_S8
   generic( INIT_00 : string :=
"0000000000000000000000000000000000000000000000000000000000000000");
   port (
       DI : in STD_LOGIC_VECTOR (7 downto 0);
       EN : in STD_ULOGIC;
       WE : in STD_ULOGIC;
       RST : in STD_ULOGIC;
       CLK : in STD_ULOGIC;
       ADDR : in STD_LOGIC_VECTOR (8 downto 0);
       DO : out STD_LOGIC_VECTOR (7 downto 0)
       );
  end component;

begin
INST_RAMB4_S4 : RAMB4_S4
  generic map (INIT_00 =>
  "1F1E1D1C1B1A191817161514131211100F0E0D0C0B0A0980706050403020100")
  port map (
       DI => DI(3 downto 0),
       EN => EN,
       WE => WE,
       RST => RST,
       CLK => CLK,
       ADDR => ADDR(9 downto 0),
       DO => DORAMB4_S4
       );

INST_RAMB4_S8 : RAMB4_S8
  generic map (INIT_00 =>
  "1F1E1D1C1B1A191817161514131211100F0E0D0C0B0A0980706050403020100")
port map (
       DI => DI(7 downto 0),
       EN => EN,
       WE => WE,
       RST => RST,
       CLK => CLK,
       ADDR => ADDR(8 downto 0),
       DO => DORAMB4_S8
       );
end struct;
```

- Synplify™

  With Synplify™ you can instantiate the RAMB* cells by using the Xilinx® family library supplied with Synplify™. The following code example illustrates instantiation of a RAMB* cell.

```
library IEEE;
use IEEE.std_logic_1164.all;
library virtex;
use virtex.components.all;
library synplify;
use synplify.attributes.all;
```

```
entity RAMB4_S8_synp is
  generic (INIT_00, INIT_01 : string :=
   "000000000000000000000000000000000000000000000000000000000000000");
  port (
        WE, EN, RST, CLK : in std_logic;
        ADDR : in std_logic_vector(8 downto 0);
        DI : in std_logic_vector(7 downto 0);
        DO : out std_logic_vector(7 downto 0)
        );
end RAMB4_S8_synp;
architecture XILINX of RAMB4_S8_synp is
  component RAMB4_S8
    port (
        WE, EN, RST, CLK : in STD_LOGIC;
        ADDR : in STD_LOGIC_VECTOR(8 downto 0);
        DI : in STD_LOGIC_VECTOR(7 downto 0);
        DO : out STD_LOGIC_VECTOR(7 downto 0)
        );
  end component;
  attribute xc_props of u1 : label is
        "INIT_00=" & INIT_00 & ",INIT_01=" & INIT_01;
  begin
    U1 : RAMB4_S8
    port map (WE => WE, EN => EN, RST => RST, CLK => CLK,
                ADDR => ADDR, DI => DI, DO => DO);
end XILINX;

library IEEE;
use IEEE.std_logic_1164.all;

entity block_ram_ex is
port (
        CLK, WE : in std_logic;
        ADDR : in std_logic_vector(8 downto 0);
        DIN : in std_logic_vector(7 downto 0);
        DOUT : out std_logic_vector(7 downto 0)
        );
end block_ram_ex;

architecture XILINX of block_ram_ex is
component RAMB4_S8_synp
  generic( INIT_00, INIT_01 : string :=
   "000000000000000000000000000000000000000000000000000000000000000");
  port (
        WE, EN, RST, CLK : in STD_LOGIC;
        ADDR : in STD_LOGIC_VECTOR(8 downto 0);
        DI : in STD_LOGIC_VECTOR(7 downto 0);
        DO : out STD_LOGIC_VECTOR(7 downto 0));
end component;
begin
  U1 : RAMB4_S8_synp
  generic map (
  INIT_00 =>
   "0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF023456789ABCDEF",
  INIT_01 =>
   "FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FDCBA9876543210")
  port map (WE => WE, EN => ’1’, RST => ’0’, CLK => CLK,
        ADDR => ADDR, DI => DIN, DO => DOUT);
end XILINX;
```

### Instantiating Block SelectRAM™ Verilog Example

The following Verilog examples show Block SelectRAM™ instantiation.

- FPGA Compiler II™

  With FPGA Compiler II™ the INIT attribute has to be set in the HDL code. See the following example.

```
module block_ram_ex (CLK, WE, ADDR, DIN, DOUT);
  input CLK, WE;
  input [8:0] ADDR;
  input [7:0] DIN;
  output [7:0] DOUT;
  RAMB4_S8 U0 (
        .WE(WE),
        .EN(1’b1),
        .RST(1’b0),
        .CLK(CLK),
        .ADDR(ADDR),
        .DI(DIN),
        .DO(DOUT)
        );
//synopsys attribute
  INIT_00
    “1F1E1D1C1B1A191817161514131211100F0E0D0C0B0A0980706050403020100”
endmodule
```

- LeonardoSpectrum™

  With LeonardoSpectrum™ the INIT attribute can be set in the HDL code or in the command line. See the following example.

```
set_attribute -instance "inst_ramb4_s4" -name
INIT_00 -type string value
    "1F1E1D1C1B1A191817161514131211100F0E0D0C0B0A0908006050403020100"
```

- LeonardoSpectrum™ block_ram_ex Verilog example

```
module block_ram_ex (CLK, WE, ADDR, DIN, DOUT);
  input CLK, WE;
  input [8:0] ADDR;
  input [7:0] DIN;
  output [7:0] DOUT;
  RAMB4_S8 U0 (
        .WE(WE),
        .EN(1’b1),
        .RST(1’b0),
        .CLK(CLK),
        .ADDR(ADDR),
        .DI(DIN),
        .DO(DOUT));
//exemplar attribute U0 INIT_00
  1F1E1D1C1B1A191817161514131211100F0E0D0C0B0A09080706050403020100
endmodule
```

- Synplicity® block_ram_ex Verilog example

```
module block_ram_ex (CLK, WE, ADDR, DIN, DOUT);
  input CLK, WE;
  input [8:0] ADDR;
  input [7:0] DIN;
  output [7:0] DOUT;
// synthesis translate_off
  defparam
  U0.INIT_00 =
256'h0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF,
  U0.INIT_01 =
256'hFEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210;
// synthesis translate_on
  RAMB4_S8 U0 (
        .WE(WE),
        .EN(1'b1),
        .RST(1'b0),
        .CLK(CLK),
        .ADDR(ADDR),
        .DI(DIN),
        .DO(DOUT))
/* synthesis
  xc_props =
  "INIT_00 =
   0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF,
  INIT_01 =
   FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210"*;
endmodule
```

- XST

  With XST, the INIT attribute must set in the HDL code. See the following example.

```
module block_ram_ex (CLK, WE, ADDR, DIN, DOUT);
  input CLK, WE;
  input [8:0] ADDR;
  input [7:0] DIN;
  output [7:0] DOUT;
  RAMB4_S8 U0 (.WE(WE), .EN(1'b1), .RST(1'b0),
        .CLK(CLK), .ADDR(ADDR), .DI(DIN), .DO(DOUT));
//synthesis attribute INIT_00 of U0 is
  “1F1E1D1C1B1A191817161514131211100F0E0D0C0B0A0980706050403020100”
endmodule
```

## Instantiating Block SelectRAM™ in Virtex™-II/II Pro/II Pro X and Spartan-3™

Virtex™-II / II Pro / II Pro X and Spartan-3™ devices provide 16384-bit data memory and 2048-bit parity memory, totaling to 18Mbit memory in each Block SelectRAM™. These RAMB16_Sn (single port) and RAMB16_Sm_Sn (dual port) blocks are configurable to various width and depth. The *Virtex-II Platform FPGA User Guide* provides VHDL and Verilog templates for Virtex™-II / II Pro / II Pro X and Spartan-3™ Block SelectRAM™ instantiations. You can also refer to the *Libraries Guide* for a more detailed component and attribute description.

### Inferring Block SelectRAM™

The following coding examples provide VHDL and Verilog coding styles for FPGA Compiler II™, LeonardoSpectrum™, Synplify™, and XST. For Virtex™, Virtex-E™ and Spartan-II™ devices, the RAMB4_Sn or RAMB4_Sm_Sn is inferred. For Virtex™-II / II Pro / II Pro X or Spartan-3™ devices, RAMB16_Sn or RAMB16_Sm_Sn is inferred.

#### Inferring Block SelectRAM™ VHDL Example

Block SelectRAM™ can be inferred by some synthesis tools. Inferred RAM must be initialized in the UCF file. Not all Block SelectRAM™ features can be inferred. Those features are pointed out in this section.

- FPGA Compiler II™

  RAM inference is not supported by FPGA Compiler II™.

- LeonardoSpectrum™

LeonardoSpectrum™ can map your memory statements in Verilog or VHDL to the Block SelectRAM™ on all Virtex™ devices. The following is a list of the details for Block SelectRAM™ in LeonardoSpectrum™.

  - Virtex™ Block SelectRAM™ is completely synchronous — both read and write operations are synchronous.
  - LeonardoSpectrum™ infers single port RAMs — RAMs with both read and write on the same address — and dual port RAMs — RAMs with separate read and write addresses.
  - Virtex™ Block SelectRAM™ supports RST (reset) and ENA (enable) pins. Currently, LeonardoSpectrum™ does not infer RAMs which use the functionality of the RST and ENA pins.
  - By default, RAMs are mapped to Block SelectRAM™ if possible. You can disable mapping to Block SelectRAM™ by setting the attribute BLOCK_RAM to false.

- LeonardoSpectrum™ VHDL example:

```
library ieee, exemplar;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ram_example1 is
  generic(data_width : integer := 8;
  address_width : integer := 8;
  mem_depth : integer : = 256);
  port (
        data : in std_logic_vector(data_width-1 downto 0);
        address : in unsigned(address_width-1 downto 0);
        we, clk : in std_logic;
        q : out std_logic_vector(data_width-1 downto 0)
        );
end ram_example1;

architecture ex1 of ram_example1 is

  type mem_type is array (mem_depth-1 downto 0)
      of std_logic_vector (data_width-1 downto 0);
  signal mem : mem_type;
  signal raddress : unsigned(address_width-1 downto 0);
  begin
  l0: process (clk, we, address)
  begin
    if (clk = ’1’ and clk’event) then
        raddress <= address;
        if (we = ’1’) then
            mem(to_integer(raddress)) <= data;
        end if;
    end if;
  end process;
  l1: process (clk, address)
  begin
    if (clk = ’1’ and clk’event) then
        q <= mem(to_integer(address));
    end if;
  end process;
end ex1;
```

- LeonardoSpectrum™ VHDL example dual port Block SelectRAM™:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity dualport_ram is
  port (
        clka : in std_logic;
        clkb : in std_logic;
        wea : in std_logic;
        addra : in std_logic_vector(4 downto 0);
        addrb : in std_logic_vector(4 downto 0);
        dia : in std_logic_vector(3 downto 0);
        dob : out std_logic_vector(3 downto 0));
end dualport_ram;

architecture dualport_ram_arch of dualport_ram is
  type ram_type is array (31 downto 0) of std_logic_vector (3 downto 0);
  signal ram : ram_type;

  attribute block_ram : boolean;
  attribute block_ram of RAM : signal is TRUE;

  begin
    write: process (clka)
    begin
      if (clka’event and clka = ’1’) then
          if (wea = ’1’) then
              ram(conv_integer(addra)) <= dia;
          end if;
      end if;
    end process write;

    read: process (clkb)
    begin
      if (clkb’event and clkb = ’1’) then
          dob <= ram(conv_integer(addrb));
      end if;
    end process read;

end dualport_ram_arch;
```

- Synplify™

  You can enable the usage of Block SelectRAM™ by setting the attribute SYN_RAMSTYLE to "block_ram". Place the attribute on the output signal driven by the inferred RAM. Remember to include the range of the output signal (bus) as part of the name.

  For example,

  ```
  define_attribute {a|dout[3:0]} syn_ramstyle "block_ram"
  ```

  The following are limitations of inferring Block SelectRAM™:

  - ENA / ENB pins currently are inaccessible. The are always tied to “1”.
  - RSTA / RSTB pins currently are inaccessible. They are always inactive.
  - Automatic inference is not yet supported. The syn_ramstyle attribute is required for inferring Block SelectRAM™.
  - Initialization of RAMs is not supported.
  - Dual port with Read-Write on a port is not supported.

- Synplify™ VHDL example

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity ram_example1 is
  generic(
        data_width : integer := 8;
        address_width : integer := 8;
        mem_depth : integer:= 256
        );
  port (
        data : in std_logic_vector(data_width-1 downto 0);
        address : in std_logic_vector(address_width-1 downto 0);
        we, clk : in std_logic;
        q : out std_logic_vector(data_width-1 downto 0)
        );
end ram_example1;

architecture rtl of ram_example1 is type mem_array is array
    (mem_depth-1 downto 0) of std_logic_vector (data_width-1 downto 0);
  signal mem : mem_array;
  attribute syn_ramstyle : string;
  attribute syn_ramstyle of mem : signal is "block_ram";
  signal raddress : std_logic_vector(address_width-1 downto 0);
    begin
    l0: process (clk)
    begin
      if (clk = ’1’ and clk’event) then
          raddress <= address;
          if (we = ’1’) then
              mem(CONV_INTEGER(address)) <= data;
          end if;
      end if;
    end process;
    q <= mem(CONV_INTEGER(raddress));
end rtl;
```

- VHDL example for Synplify™ 7.0

  In Synplify™ 7.0, the same conditions exist as with the previous release except that there is a new coding style for Block Select RAM inference in VHDL.

  The following is a Synplify™ 7.0 VHDL example

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity ram_example1 is
  generic(
        data_width : integer := 8;
        address_width : integer := 8;
        mem_depth : integer := 256);
  port(
        data : in std_logic_vector(data_width-1 downto 0);
        address : in std_logic_vector(address_width-1 downto 0);
        en, we, clk : in std_logic;
        q : out std_logic_vector(data_width-1 downto 0));
end ram_example1;

architecture rtl of ram_example1 is

  type mem_array is array (mem_depth-1 downto 0) of
        std_logic_vector (data_width-1 downto 0);
  signal mem : mem_array;
  attribute syn_ramstyle : string;
  attribute syn_ramstyle of mem : signal is "block_ram";
  signal raddress : std_logic_vector(address_width-1 downto 0);

  begin
    l0: process (clk)
    begin
      if (clk = '1' and clk'event) then
          if (we = '1') then
              mem(CONV_INTEGER(address)) <= data;
              q <= mem(CONV_INTEGER(address));
          end if;
      end if;
    end process;
end rtl;
```

- Verilog example for Synplify™ 7.0

  In Synplify™ 7.0, the same conditions exist as with the previous release except that there is a new coding style for Block Select RAM inference in Verilog.

  The following is a Synplify™ 7.0 VHDL example.

```
module sp_ram(din, addr, we, clk, dout);
  parameter data_width=16, address_width=10, mem_elements=600;
  input [data_width-1:0] din;
  input [address_width-1:0] addr;
  input rst, we, clk;
  output [data_width-1:0] dout;

reg [data_width-1:0] mem[mem_elements-1:0]
        /*synthesis syn_ramstyle = "block_ram" */;
reg [data_width-1:0] dout;
```

```
always @(posedge clk)
  begin
    if (we)
      mem[addr] <= din;
    dout <= mem[addr];
  end
endmodule
```

### Inferring Block SelectRAM™ Verilog Example

The following coding examples provide Verilog coding styles for FPGA Compiler II™, LeonardoSpectrum™, Synplify™, and XST.

- FPGA Compiler II™

  FPGA Compiler II™ does not infer RAMs. All RAMs must be instantiated via primitives or cores.

- LeonardoSpectrum™

  Refer to the VHDL example in the section above for restrictions in inferring Block SelectRAM™.

- LeonardoSpectrum™ Verilog example

```
module dualport_ram (clka, clkb, wea, addra, addrb, dia, dob);
  input clka, clkb, wea;
  input [4:0] addra, addrb;
  input [3:0] dia;
  output [3:0] dob /* synthesis syn_ramstyle="block_ram" */;
  reg [3:0] ram [31:0];
  reg [4:0] read_dpra;
  reg [3:0] dob;
// exemplar attribute ram block_ram TRUE
  always @ (posedge clka)
    begin
      if (wea) ram[addra] = dia;
    end
  always @ (posedge clkb)
    begin
      dob = ram[addrb];
    end
endmodule // dualport_ram
```

- Synplify™ Verilog example

```
module sp_ram(din, addr, we, clk, dout);
  parameter data_width=16, address_width=10, mem_elements=600;
  input [data_width-1:0] din;
  input [address_width-1:0] addr;
  input we, clk;
  output [data_width-1:0] dout;
  reg [data_width-1:0] mem[mem_elements-1:0]
        /*synthesis syn_ramstyle = "block_ram" */;
  reg [address_width - 1:0] addr_reg;
  always @(posedge clk)
    begin
        addr_reg <= addr;
          if (we)
            mem[addr] <= din;
    end
  assign dout = mem[addr_reg];
endmodule
```

## Implementing Distributed SelectRAM™

Distributed SelectRAM™ can be either instantiated or inferred. The following sections describe and give examples of both instantiating and inferring distributed SelectRAM.

The following RAM Primitives are available for instantiation.

- Static synchronous single-port RAM (RAM16x1S, RAM32x1S)

  Additional single-port RAM available for Virtex™-II / II Pro / II Pro X and Spartan-3™ devices only: RAM16X2S, RAM16X4S, RAM16X8S, RAM32X1S, RAM32X2S, RAM32X4S, RAM32X8S, RAM64X1S, RAM64X2S, and RAM128X1S.

- Static synchronous dual-port RAM (RAM16x1D, RAM32x1D)

  Additional dual-port RAM is available for Virtex™-II / II Pro / II Pro X or Spartan-3™ devices only: RAM64X1D.

For more information on distributed SelectRAM, refer to the *Libraries Guide*.

### Instantiating Distributed SelectRAM™ in VHDL

The following examples provide VHDL coding hints for FPGA Compiler II™, LeonardoSpectrum™, Synplify™ and XST.

- FPGA Compiler II™ and XST

```
-- This example shows how to create a
-- 16x4s RAM using xilinx RAM16x1S component.
library IEEE;
use IEEE.std_logic_1164.all;
-- use IEEE.std_logic_unsigned.all;
entity ram_16x4s is
  port (
       o   : out std_logic_vector(3 downto 0);
       we  : in std_logic;
       clk : in std_logic;
       d   : in std_logic_vector(3 downto 0);
       a   : in std_logic_vector(3 downto 0));
end ram_16x4s;
architecture xilinx of ram_16x4s is
  component RAM16x1S is
    port (
       O : out std_logic;
       D : in std_logic;
       A3, A2, A1, A0 : in std_logic;
       WE, WCLK : in std_logic);
  end component;
  attribute INIT: string;
  attribute INIT of U0: label is "FFFF";
  attribute INIT of U1: label is "ABCD";
  attribute INIT of U2: label is "BCDE";
  attribute INIT of U3: label is "CDEF";
  begin
  U0 : RAM16x1S
      port map (O => o(0), WE => we, WCLK => clk, D => d(0),
      A0 => a(0), A1 => a(1), A2 => a(2), A3 => a(3));
  U1 : RAM16x1S
      port map (O => o(1), WE => we, WCLK => clk, D => d(1),
        A0 => a(0), A1 => a(1), A2 => a(2), A3 => a(3));
  U2 : RAM16x1S
     port map (O => o(2), WE => we, WCLK => clk, D => d(2),
       A0 => a(0), A1 => a(1), A2 => a(2), A3 => a(3));
  U3 : RAM16x1S
     port map (O => o(3), WE => we, WCLK => clk, D => d(3),
       A0 => a(0), A1 => a(1), A2 => a(2), A3 => a(3));
end xilinx;
```

- XST

```
-- This example shows how to create a
-- 16x4s RAM using xilinx RAM16x1S component.
library IEEE;
use IEEE.std_logic_1164.all;

entity ram_16x4s is
  port (
       o    : out std_logic_vector(3 downto 0);
       we   : in std_logic;
       clk  : in std_logic;
       d    : in std_logic_vector(3 downto 0);
       a    : in std_logic_vector(3 downto 0)
       );
end ram_16x4s;

architecture xilinx of ram_16x4s is

  component RAM16x1S is
    generic (INIT : string := "0000");
    port (
       O : out std_logic;
       D : in std_logic;
       A3, A2, A1, A0 : in std_logic;
       WE, WCLK : in std_logic
       );
  end component;

  begin
    U0 : RAM16x1S
    generic map (INIT => "FFFF")
    port map (O => o(0), WE => we, WCLK => clk, D => d(0),
              A0 => a(0), A1 => a(1), A2 => a(2), A3 => a(3));

    U1 : RAM16x1S
    generic map (INIT => "ABCD")
    port map (O => o(1), WE => we, WCLK => clk, D => d(1),
              A0 => a(0), A1 => a(1), A2 => a(2), A3 => a(3));

    U2 : RAM16x1S
    generic map (INIT => "BCDE")
    port map (O => o(2), WE => we, WCLK => clk, D => d(2),
              A0 => a(0), A1 => a(1), A2 => a(2), A3 => a(3));

    U3 : RAM16x1S
    generic map (INIT => "CDEF")
    port map (O => o(3), WE => we, WCLK => clk, D => d(3),
              A0 => a(0), A1 => a(1), A2 => a(2), A3 => a(3));
end xilinx;
```

- LeonardoSpectrum™

```
-- This example shows how to create a
-- 16x4s RAM using xilinx RAM16x1S component.

library IEEE;
use IEEE.std_logic_1164.all;
entity ram_16x1s is
  generic (init_val : string := "0000");
  port (
       O : out std_logic;
       D : in std_logic;
       A3, A2, A1, A0: in std_logic;
       WE, CLK : in std_logic
       );
end ram_16x1s;
architecture xilinx of ram_16x1s is
  attribute INIT: string;
  attribute INIT of u1 : label is init_val;
  component RAM16X1S is port (
       O : out std_logic;
       D : in std_logic;
       WE : in std_logic;
       WCLK : in std_logic;
       A0 : in std_logic;
       A1 : in std_logic;
       A2 : in std_logic;
       A3 : in std_logic);
  end component;
begin
  U1 : RAM16X1S
     port map (O => O,WE => WE,WCLK => CLK,D => D,A0 => A0,
       A1 => A1,A2 => A2,A3 => A3);
end xilinx;

library IEEE;
use IEEE.std_logic_1164.all;
--use IEEE.std_logic_unsigned.all;
entity ram_16x4s is
  port (
       o : out std_logic_vector(3 downto 0);
       we : in std_logic;
       clk : in std_logic;
       d : in std_logic_vector(3 downto 0);
       a : in std_logic_vector(3 downto 0));
end ram_16x4s;
architecture xilinx of ram_16x4s is
  component ram_16x1s
    generic (init_val : string := "0000");
    port (
       O : out std_logic;
       D : in std_logic;
       A3, A2, A1, A0 : in std_logic;
       WE, CLK : in std_logic
       );
  end component;
begin
```

```
  U0 : ram_16x1s generic map ("FFFF")
    port map (O => o(0), WE => we, CLK => clk, D => d(0),
        A0 => a(0), A1 => a(1), A2 => a(2),A3 => a(3));
  U1 : ram_16x1s generic map ("ABCD")
    port map (O => o(1), WE => we, CLK => clk, D => d(1),
        A0 => a(0), A1 => a(1), A2 => a(2), A3 => a(3));
  U2 : ram_16x1s generic map ("BCDE")
    port map (O => o(2), WE => we, CLK => clk, D => d(2),
        A0 => a(0), A1 => a(1), A2 => a(2), A3 => a(3));
  U3 : ram_16x1s generic map ("CDEF")
    port map (O => o(3), WE => we, CLK => clk, D => d(3),
        A0 => a(0), A1 => a(1), A2 => a(2), A3 => a(3));
end xilinx;
```

- Synplify™

```
-- This example shows how to create a
-- 16x4s RAM using xilinx RAM16x1S component.
library IEEE;
use IEEE.std_logic_1164.all;
library virtex;
use virtex.components.all;
library synplify;
use synplify.attributes.all;

entity ram_16x1s is
  generic (init_val : string := "0000");
  port (
        O : out std_logic;
        D : in std_logic;
        A3, A2, A1, A0 : in std_logic;
        WE, CLK : in std_logic);
end ram_16x1s;

architecture xilinx of ram_16x1s is
  attribute xc_props: string;
  attribute xc_props of u1 : label is "INIT=" & init_val;

begin
  U1 : RAM16X1S
    port map (O => O, WE => WE, WCLK => CLK, D => D, A0 => A0,
        A1 => A1, A2 => A2, A3 => A3);
end xilinx;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity ram_16x4s is
  port (
        o : out std_logic_vector(3 downto 0);
        we : in std_logic;
        clk : in std_logic;
        d : in std_logic_vector(3 downto 0);
        a : in std_logic_vector(3 downto 0));
end ram_16x4s;
```

```
architecture xilinx of ram_16x4s is

  component ram_16x1s
    generic (init_val: string := "0000");
    port (
        O : out std_logic;
        D : in std_logic;
        A3, A2, A1, A0 : in std_logic;
        WE, CLK : in std_logic);
  end component;

begin
  U0 : ram_16x1s generic map ("FFFF")
    port map (O => o(0), WE => we, CLK => clk, D =>d(0), A0 => a(0),
        A1 => a(1), A2 => a(2), A3 => a(3));
  U1 : ram_16x1s generic map ("ABCD")
    port map (O => o(1), WE => we, CLK => clk, D => d(1), A0 => a(0),
        A1 => a(1), A2 => a(2), A3 => a(3));
  U2 : ram_16x1s generic map ("BCDE")
    port map (O => o(2), WE => we, CLK => clk, D => d(2), A0 => a(0),
        A1 => a(1), A2 => a(2), A3 => a(3));
  U3 : ram_16x1s generic map ("CDEF")
    port map (O => o(3), WE => we, CLK => clk, D => d(3), A0 => a(0),
        A1 => a(1), A2 => a(2), A3 => a(3));
end xilinx;
```

### Instantiating Distributed SelectRAM™ in Verilog

The following coding provides Verilog coding hints for FPGA Compiler II™, LeonardoSpectrum™, Synplify™, and XST.

- FPGA Compiler II™

```
// This example shows how to create a
// 16x4 RAM using Xilinx RAM16X1S component.
module RAM_INIT_EX1 (DATA_BUS, ADDR, WE, CLK);
  input [3:0] ADDR;
  inout [3:0] DATA_BUS;
  input WE, CLK;
  wire [3:0] DATA_OUT;
// Only for Simulation
// -- the defparam will not synthesize
// Use the defparam for RTL simulation.
// There is no defparam needed for
// Post P&R simulation.
// synopsys translate_off
  defparam RAM0.INIT="0101", RAM1.INIT="AAAA",
            RAM2.INIT="FFFF", RAM3.INIT="5555";
// synopsys translate_on
  assign DATA_BUS = !WE ? DATA_OUT : 4'hz;
// Instantiation of 4 16X1 Synchronous RAMs
// Use the xc_props attribute
// to pass the INIT property
  RAM16X1S RAM3 (
       .O (DATA_OUT[3]),.D (DATA_BUS[3]),.A3 (ADDR[3]),.A2 (ADDR[2]),
       .A1 (ADDR[1]),.A0 (ADDR[0]),.WE (WE),.WCLK (CLK)) ;
  /* synopsys attribute INIT "5555" */
  RAM16X1S RAM2 (
       .O (DATA_OUT[2]),.D (DATA_BUS[2]),.A3 (ADDR[3]),.A2 (ADDR[2]),
       .A1 (ADDR[1]),.A0 (ADDR[0]),.WE (WE),.WCLK (CLK));
  /* synopsys attribute INIT "FFFF" */
  RAM16X1S RAM1 (
       .O (DATA_OUT[1]), .D (DATA_BUS[1]), .A3 (ADDR[3]),
       .A2 (ADDR[2]), .A1 (ADDR[1]), .A0 (ADDR[0]), .WE (WE),
       .WCLK (CLK));
  /* synopsys attribute INIT "AAAA" */
  RAM16X1S RAM0 (
       .O (DATA_OUT[0]),.D (DATA_BUS[0]),.A3 (ADDR[3]),.A2 (ADDR[2]),
       .A1 (ADDR[1]), .A0 (ADDR[0]), .WE (WE), .WCLK (CLK));
  /* synopsys attribute INIT "0101" */
endmodule
module RAM16X1S (O,D,A3, A2, A1, A0, WE, WCLK);
  output O;
  input D;
  input A3;
  input A2;
  input A1;
  input A0;
  input WE;
  input WCLK;
endmodule
```

- LeonardoSpectrum™

```
// This example shows how to create a
// 16x4 RAM using Xilinx RAM16X1S component.
module RAM_INIT_EX1 (DATA_BUS, ADDR, WE, CLK);
  input [3:0] ADDR;
  inout [3:0] DATA_BUS;
  input WE, CLK;
  wire [3:0] DATA_OUT;
// Only for Simulation
// -- the defparam will not synthesize
// Use the defparam for RTL simulation.
// There is no defparam needed for
// Post P&R simulation.
// exemplar translate_off
  defparam RAM0.INIT="0101", RAM1.INIT="AAAA", RAM2.INIT="FFFF",
       RAM3.INIT="5555";
// exemplar translate_on
  assign DATA_BUS = !WE ? DATA_OUT : 4’hz;
// Instantiation of 4 16X1 Synchronous RAMs
  RAM16X1S RAM3 (
     .O (DATA_OUT[3]),.D (DATA_BUS[3]),.A3 (ADDR[3]),.A2 (ADDR[2]),
       .A1 (ADDR[1]), .A0 (ADDR[0]), .WE (WE), .WCLK (CLK))
  /* exemplar attribute RAM3 INIT 5555 */;
  RAM16X1S RAM2 (
     .O (DATA_OUT[2]), .D (DATA_BUS[2]), .A3 (ADDR[3]) ,.A2 (ADDR[2]),
     .A1 (ADDR[1]), .A0 (ADDR[0]), .WE (WE), .WCLK (CLK))
  /* exemplar attribute RAM2 INIT FFFF */;
  RAM16X1S RAM1 (
     .O (DATA_OUT[1]), .D (DATA_BUS[1]), .A3 (ADDR[3]), .A2 (ADDR[2]),
     .A1 (ADDR[1]), .A0 (ADDR[0]), .WE (WE), .WCLK (CLK))
  /* exemplar attribute RAM1 INIT AAAA */;
  RAM16X1S RAM0 (
     .O (DATA_OUT[0]), .D (DATA_BUS[0]), .A3 (ADDR[3]), .A2 (ADDR[2]),
     .A1 (ADDR[1]), .A0 (ADDR[0]), .WE (WE), .WCLK (CLK))
  /* exemplar attribute RAM0 INIT 0101 */;
endmodule
module RAM16X1S (O,D,A3, A2, A1, A0, WE, WCLK);
  output O;
  input D;
  input A3;
  input A2;
  input A1;
  input A0;
  input WE;
  input WCLK;
endmodule
```

- Synplify™

```
// This example shows how to create a
// 16x4 RAM using Xilinx RAM16X1S component.
‘include "virtex.v"
module RAM_INIT_EX1 (DATA_BUS, ADDR, WE, CLK);
  input [3:0] ADDR;
  inout [3:0] DATA_BUS;
  input WE, CLK;
  wire [3:0] DATA_OUT;
// Only for Simulation
// -- the defparam will not synthesize
// Use the defparam for RTL simulation.
// There is no defparam needed for
// Post P&R simulation.
// synthesis translate_off
  defparam RAM0.INIT="0101", RAM1.INIT="AAAA",RAM2.INIT="FFFF",
      RAM3.INIT="5555";
// synthesis translate_on
  assign DATA_BUS = !WE ? DATA_OUT : 4’hz;
// Instantiation of 4 16X1 Synchronous RAMs
// Use the xc_props attribute to pass the INIT
// property
  RAM16X1S RAM3 (
      .O (DATA_OUT[3]), .D (DATA_BUS[3]), .A3 (ADDR[3]), .A2 (ADDR[2]),
      .A1 (ADDR[1]), .A0 (ADDR[0]), .WE (WE), .WCLK (CLK))
  /* synthesis xc_props="INIT=5555" */;
  RAM16X1S RAM2 (
      .O (DATA_OUT[2]), .D (DATA_BUS[2]), .A3 (ADDR[3]), .A2 (ADDR[2]),
      .A1 (ADDR[1]), .A0 (ADDR[0]), .WE (WE), .WCLK (CLK))
  /* synthesis xc_props="INIT=FFFF" */;
  RAM16X1S RAM1 (
      .O (DATA_OUT[1]), .D (DATA_BUS[1]), .A3 (ADDR[3]), .A2 (ADDR[2]),
      .A1 (ADDR[1]), .A0 (ADDR[0]), .WE (WE), .WCLK (CLK))
  /* synthesis xc_props="INIT=AAAA" */;
  RAM16X1S RAM0 (
      .O (DATA_OUT[0]), .D (DATA_BUS[0]), .A3 (ADDR[3]), .A2 (ADDR[2]),
      .A1 (ADDR[1]), .A0 (ADDR[0]), .WE (WE), .WCLK (CLK))
  /* synthesis xc_props="INIT=0101" */;
endmodule
```

- XST

```
// This example shows how to create a
// 16x4 RAM using Xilinx RAM16X1S component.
module RAM_INIT_EX1 (DATA_BUS, ADDR, WE, CLK);
  input [3:0] ADDR;
  inout [3:0] DATA_BUS;
  input WE, CLK;
  wire [3:0] DATA_OUT;
// Only for Simulation --
// the defparam will not synthesize
// Use the defparam for RTL simulation.
// There is no defparam needed for
// Post P&R simulation.
// synthesis translate_off
  defparam RAM0.INIT="0101", RAM1.INIT="AAAA", RAM2.INIT="FFFF",
        RAM3.INIT="5555";
// synthesis translate_on
  assign DATA_BUS = !WE ? DATA_OUT : 4'hz;
// Instantiation of 4 16X1 Synchronous RAMs
// Use the xc_props attribute to
// pass the INIT property

// synthesis attribute INIT of RAM2 is "FFFF"
// synthesis attribute INIT of RAM1 is "AAAA"
// synthesis attribute INIT of RAM0 is "0101"

  RAM16X1S RAM3 (
      .O (DATA_OUT[3]), .D (DATA_BUS[3]), .A3 (ADDR[3]), .A2 (ADDR[2]),
      .A1 (ADDR[1]), .A0 (ADDR[0]), .WE (WE), .WCLK (CLK))
  RAM16X1S RAM2 (
      .O (DATA_OUT[2]), .D (DATA_BUS[2]), .A3 (ADDR[3]), .A2 (ADDR[2]),
      .A1 (ADDR[1]), .A0 (ADDR[0]), .WE (WE), .WCLK (CLK));

  RAM16X1S RAM1 (
      .O (DATA_OUT[1]), .D (DATA_BUS[1]), .A3 (ADDR[3]), .A2 (ADDR[2]),
      .A1 (ADDR[1]), .A0 (ADDR[0]), .WE (WE), .WCLK (CLK));

  RAM16X1S RAM0 (
      .O (DATA_OUT[0]), .D (DATA_BUS[0]), .A3 (ADDR[3]), .A2 (ADDR[2]),
      .A1 (ADDR[1]), .A0 (ADDR[0]), .WE (WE), .WCLK (CLK));
endmodule
```

### Inferring Distributed SelectRAM™ in VHDL

The following coding examples provide VHDL and Verilog coding styles for FPGA Compiler II™, LeonardoSpectrum™, Synplify™, and XST.

- FPGA Compiler II™

  FPGA Compiler II™ does not infer RAMs.

- LeonardoSpectrum™, Synplify™, and XST

  The following is a 32x8 (32 words by 8 bits per word) synchronous, dual-port RAM example.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity ram_32x8d_infer is
  generic(
        d_width : integer := 8;
        addr_width : integer := 5;
        mem_depth : integer := 32
        );
  port (
        o : out STD_LOGIC_VECTOR(d_width - 1 downto 0);
        we, clk : in STD_LOGIC;
        d : in STD_LOGIC_VECTOR(d_width - 1 downto 0);
        raddr, waddr : in STD_LOGIC_VECTOR(addr_width - 1 downto 0));
end ram_32x8d_infer;

architecture xilinx of ram_32x8d_infer is
  type mem_type is array (
        mem_depth - 1 downto 0) of
        STD_LOGIC_VECTOR (d_width - 1 downto 0);
  signal mem : mem_type;
begin
  process(clk, we, waddr)
  begin
      if (rising_edge(clk)) then
          if (we = '1') then
              mem(conv_integer(waddr)) <= d;
          end if;
      end if;
  end process;
  process(raddr)
  begin
    o <= mem(conv_integer(raddr));
  end process;
end xilinx;
```

- The following is a 32x8 (32 words by 8 bits per word) synchronous, single-port RAM example.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity ram_32x8s_infer is
  generic(
        d_width : integer := 8;
        addr_width : integer := 5;
        mem_depth : integer := 32
        );
  port (
        o : out STD_LOGIC_VECTOR(d_width - 1 downto 0);
        we, wclk : in STD_LOGIC;
        d : in STD_LOGIC_VECTOR(d_width - 1 downto 0);
        addr : in STD_LOGIC_VECTOR(addr_width - 1 downto 0));
end ram_32x8s_infer;
```

```
architecture xilinx of ram_32x8s_infer is
        type mem_type is array (mem_depth - 1 downto 0)
        of STD_LOGIC_VECTOR (d_width - 1 downto 0);
  signal mem : mem_type;
begin
  process(wclk, we, addr)
  begin
    if (rising_edge(wclk)) then
        if (we = ’1’) then
            mem(conv_integer(addr)) <= d;
        end if;
    end if;
  end process;
  o <= mem(conv_integer(addr));
end xilinx;
```

### Inferring Distributed SelectRAM™ in Verilog

The following coding examples provide Verilog coding hints for FPGA Compiler II™, Synplify™, LeonardoSpectrum™, and XST.

- FPGA Compiler II™

  FPGA Compiler II™ does not infer RAMs.

- LeonardoSpectrum™, Synplify™, and XST

  The following is a 32x8 (32 words by 8 bits per word) synchronous, dual-port RAM example.

```
module ram_32x8d_infer (o, we, d, raddr, waddr, clk);
  parameter d_width = 8, addr_width = 5;
  output [d_width - 1:0] o;
  input we, clk;
  input [d_width - 1:0] d;
  input [addr_width - 1:0] raddr, waddr;

  reg [d_width - 1:0] o;
  reg [d_width - 1:0] mem [(1 << addr_width) 1:0];

  always @(posedge clk)
    if (we)
        mem[waddr] = d;

  always @(raddr)
    o = mem[raddr];
endmodule
```

  The following is a 32x8 (32 words by 8 bits per word) synchronous, single-port RAM example.

```
module ram_32x8s_infer (o, we, d, addr, wclk);
  parameter d_width = 8, addr_width = 5;
  output [d_width - 1:0] o;
  input we, wclk;
  input [d_width - 1:0] d;
  input [addr_width - 1:0] addr;

  reg [d_width - 1:0] mem [(1 << addr_width) 1:0];
  always @(posedge wclk)
    if (we)
        mem[addr] = d;
  assign o = mem[addr];
endmodule
```

## Implementing ROMs

ROMs can be implemented as follows.

- Use RTL descriptions of ROMs
- Instantiate 16x1 and 32x1 ROM primitives

Following are RTL VHDL and Verilog ROM coding examples.

### RTL Description of a Distributed ROM VHDL Example

***Note:*** LeonardoSpectrum™ does not infer ROM.

Use the following coding example for FPGA Compiler II™, Synplify™, and XST.

```
--
--  Behavioral 16x4 ROM Example
--           rom_rtl.vhd
--
library IEEE;
use IEEE.std_logic_1164.all;

entity rom_rtl is
  port (
        ADDR : in INTEGER range 0 to 15;
        DATA : out STD_LOGIC_VECTOR (3 downto 0)
        );
end rom_rtl;

architecture XILINX of rom_rtl is
  subtype ROM_WORD is STD_LOGIC_VECTOR (3 downto 0);
  type ROM_TABLE is array (0 to 15) of ROM_WORD;
  constant ROM : ROM_TABLE := ROM_TABLE'(
        ROM_WORD'("0000"),
        ROM_WORD'("0001"),
        ROM_WORD'("0010"),
        ROM_WORD'("0100"),
        ROM_WORD'("1000"),
        ROM_WORD'("1100"),
        ROM_WORD'("1010"),
        ROM_WORD'("1001"),
        ROM_WORD'("1001"),
        ROM_WORD'("1010"),
        ROM_WORD'("1100"),
        ROM_WORD'("1001"),
        ROM_WORD'("1001"),
        ROM_WORD'("1101"),
        ROM_WORD'("1011"),
        ROM_WORD'("1111")
        );
begin
  DATA <= ROM(ADDR);    -- Read from the ROM
end XILINX;
```

## RTL Description of a Distributed ROM Verilog Example

***Note:*** LeonardoSpectrum™ does not infer ROM.

Use the following coding example for FPGA Compiler II™, Synplify™, and XST.

```
/*
 * ROM_RTL.V
 * Behavioral Example of 16x4 ROM
 */

module rom_rtl(ADDR, DATA);
  input [3:0] ADDR;
  output [3:0] DATA;
  reg [3:0] DATA;

// A memory is implemented
// using a case statement

  always @(ADDR)
  begin
    case (ADDR)
      4’b0000 : DATA = 4’b0000 ;
      4’b0001 : DATA = 4’b0001 ;
      4’b0010 : DATA = 4’b0010 ;
      4’b0011 : DATA = 4’b0100 ;
      4’b0100 : DATA = 4’b1000 ;
      4’b0101 : DATA = 4’b1000 ;
      4’b0110 : DATA = 4’b1100 ;
      4’b0111 : DATA = 4’b1010 ;
      4’b1000 : DATA = 4’b1001 ;
      4’b1001 : DATA = 4’b1001 ;
      4’b1010 : DATA = 4’b1010 ;
      4’b1011 : DATA = 4’b1100 ;
      4’b1100 : DATA = 4’b1001 ;
      4’b1101 : DATA = 4’b1001 ;
      4’b1110 : DATA = 4’b1101 ;
      4’b1111 : DATA = 4’b1111 ;
    endcase
  end
endmodule
```

With the VHDL and Verilog examples above, synthesis tools create ROMs using function generators (LUTs and MUXFs) or the ROM primitives.

Another method for implementing ROMs is to instantiate the 16x1 or 32x1 ROM primitives. To define the ROM value, use the Set Attribute or equivalent command to set the INIT property on the ROM component.

***Note:*** Refer to your synthesis tool documentation for the correct syntax.

This type of command writes the ROM contents to the netlist file so the Xilinx® tools can initialize the ROM. The INIT value should be specified in hexadecimal values. See the VHDL and Verilog RAM examples in the following section for examples of this property using a RAM primitive.

## Implementing ROMs Using Block SelectRAM™

FPGA Compiler II™, LeonardoSpectrum™ and Synplify™ can infer ROM using Block SelectRAM™.

FPGA Compiler II™:

FPGA Compiler II™ can infer ROMs using Block SelectRAM™ instead of LUTs for Virtex™, Virtex-E™, Virtex-II™, Virtex-II Pro™, Virtex-II Pro X™ and Spartan-3™ in the following cases:

- The inference is synchronous.
- For Virtex™ and Virtex-E™, Block SelectRAM™ are used to infer ROM when the address line is at least ten bits, and the data line is three bits or greater. Also, Block SelectRAM™ is used when the address line is 11 or 12 bits; no minimum data width is required.
- For Virtex-II™, Virtex-II Pro™, Virtex-II Pro X™, and Spartan-3™ Block SelectRAM™ is used to infer ROM if the address line is between 10 and 14 bits; no minimum data width is required.

LeonardoSpectrum™:

- In LeonardoSpectrum™, synchronous ROMs with address widths greater than eight bits are automatically mapped to Block SelectRAM™.
- Asynchronous ROMs and synchronous ROMs (with address widths less than eight bits) are automatically mapped to distributed SelectRAM™.

Synplify™:

Synplify™ can infer ROMs using Block SelectRAM™ instead of LUTs for Virtex™, Virtex-E™, Virtex-II™, Virtex-II Pro™, Virtex-II Pro X™ and Spartan-3™ in the following cases:

- For Virtex™ and Virtex-E™, the address line must be between 8 and 12 bits.
- For Virtex™-II, Virtex-II Pro™, Virtex-II Pro X™ or Spartan-3™, the address line must be between 9 and 14 bits.
- The address lines must be registered with a simple flip-flop (no resets or enables, etc.) or the ROM output can be registered with enables or sets/resets. However, you cannot use both sets/resets and enables. The flip-flops' sets/resets can be either synchronous or asynchronous. In the case where asynchronous sets/resets are used, Synplify™ creates registers with the sets/resets and then either AND or OR these registers with the output of the Block RAM.

### RTL Description of a ROM VHDL Example Using Block SelectRAM™

Following is some incomplete VHDL that demonstrates the above inference rules.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity rom_rtl is
  port (
        ADDR : in INTEGER range 0 to 1023;
        CLK  : in std_logic;
        DATA : out STD_LOGIC_VECTOR (3 downto 0)
        );
end rom_rtl;
```

```
architecture XILINX of rom_rtl is

  subtype ROM_WORD is STD_LOGIC_VECTOR (3 downto 0);
  type ROM_TABLE is array (0 to 1023) of ROM_WORD;
  constant ROM : ROM_TABLE := ROM_TABLE'(
       ROM_WORD'("0000"),
       ROM_WORD'("0001"),
       ROM_WORD'("0010"),
       ROM_WORD'("0100"),
       ROM_WORD'("1000"),
       ROM_WORD'("1100"),
       ROM_WORD'("1010"),
       ROM_WORD'("1001"),
       ROM_WORD'("1001"),
       ROM_WORD'("1010"),
       ROM_WORD'("1100"),
       ROM_WORD'("1001"),
       ROM_WORD'("1001"),
       ROM_WORD'("1101"),
       ROM_WORD'("1011"),
       ROM_WORD'("1111")
    :
    :
    :
       );
  begin
  process (CLK) begin
    if clk'event and clk = '1' then
       DATA <= ROM(ADDR);     -- Read from the ROM
    end if;
  end process;
end XILINX;
```

## RTL Description of a ROM Verilog Example using Block SelectRAM™

Following is some incomplete Verilog that demonstrates the above inference rules:

```
/*
* This code is incomplete but demonstrates the
* rules for inferring Block RAM for ROMs
* ROM_RTL.V
* Block RAM ROM Example
*/
module rom_rtl(ADDR, CLK, DATA) ;
  input [9:0] ADDR ;
  input CLK ;
  output [3:0] DATA ;
  reg [3:0] DATA ;
```

```
// A memory is implemented
// using a case statement

  always @(posedge CLK)
  begin
    case (ADDR)
        9'b000000000 : DATA = 4'b0000 ;
        9'b000000001 : DATA = 4'b0001 ;
        9'b000000010 : DATA = 4'b0010 ;
        9'b000000011 : DATA = 4'b0100 ;
        9'b000000100 : DATA = 4'b1000 ;
        9'b000000101 : DATA = 4'b1000 ;
        9'b000000110 : DATA = 4'b1100 ;
        9'b000000111 : DATA = 4'b1010 ;
        9'b000001000 : DATA = 4'b1001 ;
        9'b000001001 : DATA = 4'b1001 ;
        9'b000001010 : DATA = 4'b1010 ;
        9'b000001011 : DATA = 4'b1100 ;
        9'b000001100 : DATA = 4'b1001 ;
        9'b000001101 : DATA = 4'b1001 ;
        9'b000001110 : DATA = 4'b1101 ;
        9'b000001111 : DATA = 4'b1111 ;
               :
               :
               :
    endcase
  end
endmodule
```

## Implementing FIFO

FIFO can be implemented with FPGA RAMs. Xilinx® provides several Application Notes describing the use of FIFO when implementing FPGAs. Please refer to the following Xilinx® Application Notes for more information:

- Xilinx® XAPP175: *"High Speed FIFOs in Spartan-II FPGAs"*, application note, v1.0 (11/99)
- Xilinx® XAPP131: *"170MHz FIFOs using the Virtex Block SelectRAM Feature"*, v 1.2 (9/99)

## Implementing CAM

Content Addressable Memory (CAM) or associative memory is a storage device which can be addressed by its own contents.

Xilinx® provides several Application Notes describing CAM designs in Virtex™ FPGAs. Please refer to the following Xilinx® Application Notes for more information:

- XAPP201: *"An Overview of Multiple CAM Designs in Virtex Family Devices"* v 1.1(9/99)
- XAPP202: *"Content Addressable Memory (CAM) in ATM Applications"* v 1.1 (9/99)
- XAPP203: *"Designing Flexible, Fast CAMs with Virtex Family FPGAs"* v 1.1 (9/99)
- XAPP204: *"Using Block SelectRAM for High-Performance Read/Write CAMs"* v1.1 (10/99)

## Using CORE Generator™ to Implement Memory

If you must instantiate memory, use the CORE Generator™ to create a memory module larger than 32X1 (16X1 for Dual Port). Implementing memory with the CORE Generator™ is similar to implementing any module with CORE Generator™ except for defining the memory initialization file. Please reference the memory module data sheets that come with every CORE Generator™ module for specific information on the initialization file.

# Implementing Shift Registers (Virtex™/E/II/II Pro/II Pro X and Spartan™-II/3)

The SRL16 is a very efficient way to create shift registers without using up flip-flop resources. You can create shift registers that vary in length from one to sixteen bits. The SRL16 is a shift register look up table (LUT) whose inputs (A3, A2, A1,A0) determine the length of the shift register. The shift register may be of a fixed, static length or it may be dynamically adjusted. The shift register LUT contents are initialized by assigning a four-digit hexadecimal number to an INIT attribute. The first, or the left-most, hexadecimal digit is the most significant bit. If an INIT value is not specified, it defaults to a value of four zeros (0000) so that the shift register LUT is cleared during configuration. The data (D) is loaded into the first bit of the shift register during the Low-to-High clock (CLK) transition. During subsequent Low-to-High clock transitions data is shifted to the next highest bit position as new data is loaded. The data appears on the Q output when the shift register length determined by the address inputs is reached.

The Static Length Mode of SRL16 implements any shift register length from 1 to 16 bits in one LUT. Shift register length is (N+1) where N is the input address. Synthesis tools implement longer shift registers with multiple SRL16 and additional combinatorial logic for multiplexing.

In Virtex™-II / II Pro / II Pro X or Spartan-3™ devices, additional cascading shift register LUTs (SRLC16) are available. SRLC16 supports synchronous shift-out output of the last (16th) bit. This output has a dedicated connection to the input of the next SRLC16 inside the CLB. With four slices and dedicated multiplexers (MUXF5, MUXF6, and so forth) available in one Virtex™-II / II Pro / II Pro X or Spartan-3™ CLB, up to a 128-bit shift register can be implemented effectively using SRLC16. Synthesis tools, Synplify™ 7.1, LeonardoSpectrum™ 2002a, and XST can infer the SRLC16. For more information, please refer to the *Virtex-II Platform FPGA User Guide*.

Dynamic Length Mode can be implemented using SRL16 or SRLC16. Each time a new address is applied to the 4-input address pins, the new bit position value is available on the Q output after the time delay to access the LUT. LeonardoSpectrum™, Synplify™, and XST can infer a shift register component. A coding example for a dynamic SRL is included following the SRL16 inference example.

## Inferring SRL16 in VHDL

- FPGA Compiler II™, LeonardoSpectrum™, Synplify™, and XST

```
-- VHDL example design of SRL16
-- inference for Virtex
-- This design infer 16 SRL16
-- with 16 pipeline delay
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity pipeline_delay is
  generic (
        cycle : integer := 16;
        width :integer := 16
        );
  port (
        input :in std_logic_vector(width - 1 downto 0);
        clk :in std_logic;
        output :out std_logic_vector(width - 1 downto 0)
        );
end pipeline_delay;
architecture behav of pipeline_delay is
  type my_type is array (0 to cycle -1) of
        std_logic_vector(width -1 downto 0);
  signal int_sig :my_type;

begin
  main : process (clk)
    begin
      if clk'event and clk = '1' then
          int_sig <= input & int_sig(0 to cycle - 2);
      end if;
  end process main;
  output <= int_sig(cycle -1);
end behav;
```

## Inferring SRL16 in Verilog

Use the following coding example for FPGA Compiler II™, LeonardoSpectrum™, Synplify™, and XST.

- FPGA Compiler II™, LeonardoSpectrum™, Synplify™, and XST

```
// Verilog Example SRL
//This design infer 3 SRL16 with 4 pipeline delay
module srle_example (clk, enable, data_in, result);
  parameter cycle=4;
  parameter width = 3;
  input clk, enable;
  input [0:width] data_in;
  output [0:width] result;
  reg [0:width-1] shift [cycle-1:0];
  integer i;
  always @(posedge clk)
  begin
    if (enable == 1)
      begin
        for (i = (cycle-1);i >0; i=i-1)
            shift[i] = shift[i-1];
            shift[0] = data_in;
      end
  end
  assign result = shift[cycle-1];
endmodule
```

## Inferring Dynamic SRL16 in VHDL

- LeonardoSpectrum™, Synplify™ and XST

```
library IEEE;
use IEEE.std_logic_1164.all;

entity srltest is
  port (
        inData : std_logic_vector(7 downto 0);
        clk, en : in std_logic;
        outStage : in integer range 3 downto 0;
        outData : out std_logic_vector(7 downto 0));
end srltest;

architecture rtl of srltest is

  type dataAryType is array(3 downto 0) of std_logic_vector(7 downto 0);
  signal regBank : dataAryType;

begin
  outData <= regBank(outStage);
  process(clk, inData) begin

    if (clk’event and clk = ’1’) then
      if (en=’1’) then
        regBank <= (regBank(2 downto 0) & inData);
      end if;
    end if;
  end process;
end rtl;
```

## Inferring Dynamic SRL16 in Verilog

- LeonardoSpectrum™, Synplify™ and XST

```
module test_srl(clk, enable, dataIn, result, addr);

  input clk, enable;
  input [3:0] dataIn;
  input [3:0] addr;
  output [3:0] result;

  reg [3:0] regBank[15:0];
  integer i;
```

```
      always @(posedge clk)
        begin
        if (enable == 1)
            begin
              for (i=15; i>0; i=i-1)
                  begin
                    regBank[i] <= regBank[i-1];
                  end
              regBank[0] <= dataIn;
            end
        end
      assign result = regBank[addr];
    endmodule
```

### Implementing LFSR

The SRL (Shift Register LUT) implements very efficient shift registers and can be used to implement Linear Feedback Shift Registers. See the XAPP210 Application Note for a description of the implementation of Linear Feedback Shift Registers (LFSR) using the Virtex™ SRL macro. One half of a CLB can be configured to implement a 15-bit LFSR, one CLB can implement a 52-bit LFSR, and with two CLBs a 118-bit LFSR is implemented.

## Implementing Multiplexers

A 4-to-1 multiplexer can be efficiently implemented in a single Virtex™/E/II/II Pro/ II Pro X and Spartan™-II/3 family slice. The six input signals (four inputs, two select lines) use a combination of two LUTs and MUXF5 available in every slice. Up to 9 input functions can be implemented with this configuration.

In the Virtex™/E and Spartan-II™ families, larger multiplexers can be implemented using two adjacent slices in one CLB with its dedicated MUXF5s and a MUXF6.

Virtex™-II/II Pro/II Pro X and Spartan-3™ slices contain dedicated two-input multiplexers (one MUXF5 and one MUXFX per slice). MUXF5 is used to combine two LUTs. MUXFX can be used as MUXF6, MUXF7, and MUXF8 to combine 4, 8, and 16 LUTs, respectively. Please refer to the *Virtex-II Platform FPGA User Guide* for more information on designing large multiplexers in Virtex™-II/II Pro/II Pro X or Spartan-3™. This book can be found on the Xilinx® website at http://www.xilinx.com.

In addition, you can use internal tristate buffers (BUFTs) to implement large multiplexers. Large multiplexers built with BUFTs have the following advantages.

- Can vary in width with only minimal impact on area and delay
- Can have as many inputs as there are tristate buffers per horizontal longline in the target device
- Have one-hot encoded selector inputs

This last point is illustrated in the following VHDL and Verilog designs of a 5-to-1 multiplexer built with gates. Typically, the gate version of this multiplexer has binary encoded selector inputs and requires three select inputs (SEL<2:0>). The schematic representation of this design is shown in Figure 4-5, page 186.

Some synthesis tools include commands that allow you to switch between multiplexers with gates or with tristates. Check with your synthesis vendor for more information.

The VHDL and Verilog designs provided at the end of this section show a 5-to-1 multiplexer built with tristate buffers. The tristate buffer version of this multiplexer has one-hot encoded selector inputs and requires five select inputs (SEL<4:0>). The schematic representation of these designs is shown in Figure 4-6, page 187.

## Mux Implemented with Gates VHDL Example

The following example shows a MUX implemented with Gates.

```
-- MUX_GATE.VHD
-- 5-to-1 Mux Implemented in Gates

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity mux_gate is
  port (
        SEL: in STD_LOGIC_VECTOR (2 downto 0);
        A,B,C,D,E: in STD_LOGIC;
        SIG: out STD_LOGIC
        );
end mux_gate;

architecture RTL of mux_gate is begin
    SEL_PROCESS: process (SEL,A,B,C,D,E)
    begin
      case SEL is
        when "000" => SIG <= A;
        when "001" => SIG <= B;
        when "010" => SIG <= C;
        when "011" => SIG <= D;
        when others => SIG <= E;
      end case;
  end process SEL_PROCESS;
end RTL;
```

## Mux Implemented with Gates Verilog Example

The following example shows a MUX implemented with Gates.

```
// mux_gate.v
// 5-to-1 Mux Implemented in Gates

module mux_gate(SEL, A, B, C, D, E, SIG);
  input [2:0] SEL;
  input A, B, C, D, E;
  output SIG;
  reg SIG;
```

```
always @(SEL, A, B, C, D, E)
  begin
    case (SEL)
      3’b000 : SIG = A;
      3’b001 : SIG = B;
      3’b010 : SIG = C;
      3’b011 : SIG = D;
      default : SIG = E;
    endcase
  end
```

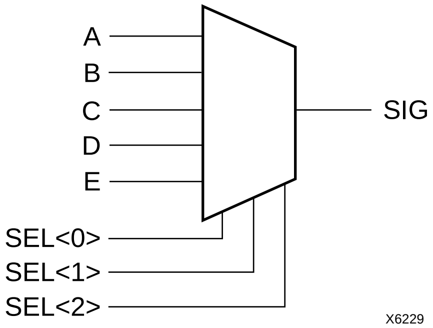


*Figure 4-5:* **5-to-1 MUX Implemented with Gates**

## Wide MUX Mapped to MUXFs

Synthesis tools use MUXF5 and MUXF6, and for Virtex-II™, Virtex-II Pro™, Virtex-II Pro X™, and Spartan-3™ use MUXF7 and MUXF8 to implement wide multiplexers. These MUXes can, respectively, be used to create a 5, 6, 7 or 8 input function or a 4-to-1, 8-to-1, 16-to-1 or a 32-to-1 multiplexer.

## Mux Implemented with BUFTs VHDL Example

The following example shows a MUX implemented with BUFTs.

```
-- MUX_TBUF.VHD
-- 5-to-1 Mux Implemented in 3-State Buffers
-- May 2001

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity mux_tbuf is
  port (
       SEL: in STD_LOGIC_VECTOR (4 downto 0);
       A,B,C,D,E: in STD_LOGIC;
       SIG: out STD_LOGIC
       );
end mux_tbuf;
```

```
architecture RTL of mux_tbuf is
begin

  SIG <= A when (SEL(0)='0') else 'Z';
  SIG <= B when (SEL(1)='0') else 'Z';
  SIG <= C when (SEL(2)='0') else 'Z';
  SIG <= D when (SEL(3)='0') else 'Z';
  SIG <= E when (SEL(4)='0') else 'Z';
end RTL;
```

## Mux Implemented with BUFTs Verilog Example

The following example shows a MUX implemented with BUFTs.

```
/* MUX_TBUF.V */
module mux_tbuf (A,B,C,D,E,SEL,SIG);
  input A,B,C,D,E;
  input [4:0] SEL;
  output SIG;

  assign SIG = (SEL[0]==1'b0) ? A : 1'bz;
  assign SIG = (SEL[1]==1'b0) ? B : 1'bz;
  assign SIG = (SEL[2]==1'b0) ? C : 1'bz;
  assign SIG = (SEL[3]==1'b0) ? D : 1'bz;
  assign SIG = (SEL[4]==1'b0) ? E : 1'bz;
endmodule
```

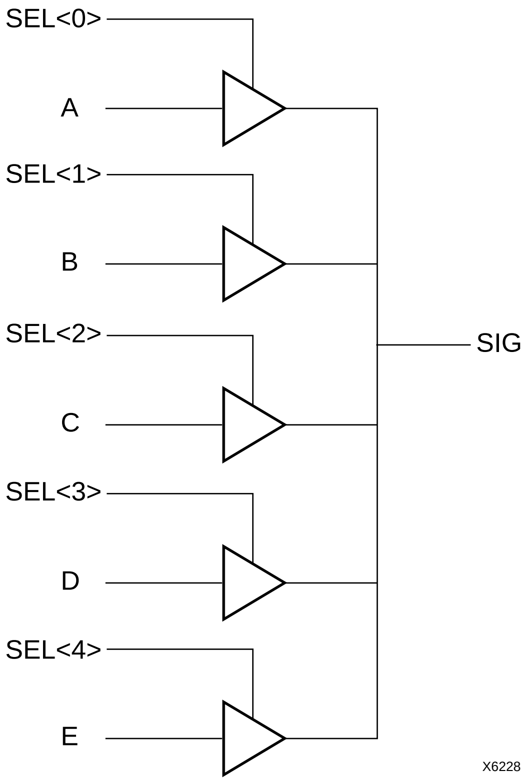


*Figure 4-6:* **5-to-1 MUX Implemented with BUFTs**

# Using Pipelining

You can use pipelining to dramatically improve device performance. Pipelining increases performance by restructuring long data paths with several levels of logic and breaking it up over multiple clock cycles. This method allows a faster clock cycle and, as a result, an increased data throughput at the expense of added data latency. Because the Xilinx® FPGA devices are register-rich, this is usually an advantageous structure for FPGA designs because the pipeline is created at no cost in terms of device resources. Because data is now on a multi-cycle path, special considerations must be used for the rest of your design to account for the added path latency. You must also be careful when defining timing specifications for these paths.

Some synthesis tools have limited capability for constraining multi-cycle paths or translating these constraints to Xilinx® implementation constraints. Check your synthesis tool documentation for information on multi-cycle paths. If your tool cannot translate the constraint but can synthesize to a multi-cycle path, you can add the constraint to the UCF file.

## Before Pipelining

In the following example, the clock speed is limited by the clock-to out-time of the source flip-flop; the logic delay through four levels of logic; the routing associated with the four function generators; and the setup time of the destination register

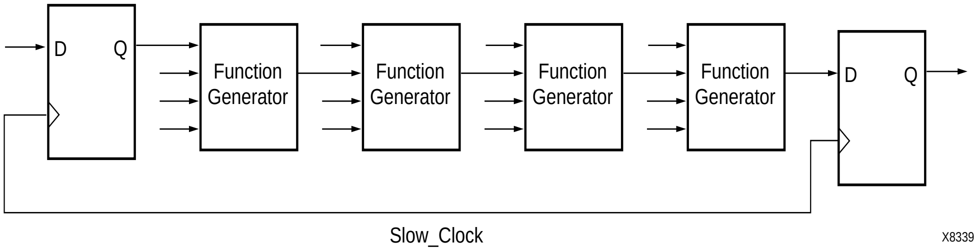


*Figure 4-7:* **Before Pipelining**

## After Pipelining

This is an example of the same data path in the previous example after pipelining. Because the flip-flop is contained in the same CLB as the function generator, the clock speed is limited by the clock-to-out time of the source flip-flop; the logic delay through one level of logic; one routing delay; and the setup time of the destination register. In this example, the system clock runs much faster than in the previous example.

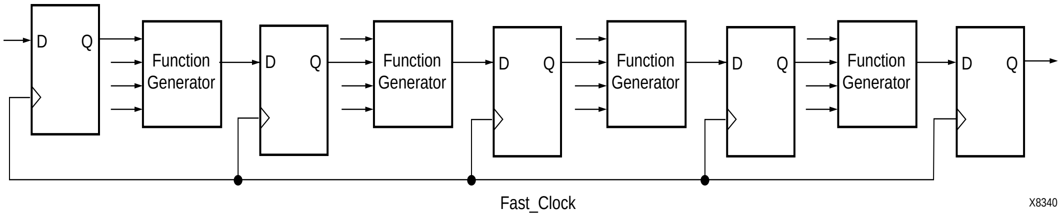


*Figure 4-8:* **After Pipelining**

# Design Hierarchy

HDL designs can either be synthesized as a flat module or as many small modules. Each methodology has its advantages and disadvantages, but as higher density FPGAs are created, the advantages of hierarchical designs outweigh any disadvantages.

Advantages to building hierarchical designs are as follows.

- Easier and faster verification / simulation
- Allows several engineers to work on one design at the same time
- Speeds up design compilation
- Reduces design time by allowing design module re-use for this and future designs.
- Allows you to produce designs that are easier to understand
- Allows you to efficiently manage the design flow

Disadvantages to building hierarchical designs are as follows.

- Design mapping into the FPGA may not be as optimal across hierarchical boundaries; this can cause lesser device utilization and decreased design performance
- Design file revision control becomes more difficult
- Designs become more verbose

Most of the disadvantages listed above can be overcome with careful design consideration when choosing the design hierarchy.

## Using Synthesis Tools with Hierarchical Designs

By effectively partitioning your designs, you can significantly reduce compile time and improve synthesis results. Here are some recommendations for partitioning your designs.

### Restrict Shared Resources to Same Hierarchy Level

Resources that can be shared should be on the same level of hierarchy. If these resources are not on the same level of hierarchy, the synthesis tool cannot determine if these resources should be shared.

### Compile Multiple Instances Together

You may want to compile multiple occurrences of the same instance together to reduce the gate count. However, to increase design speed, do not compile a module in a critical path with other instances.

### Restrict Related Combinatorial Logic to Same Hierarchy Level

Keep related combinatorial logic in the same hierarchical level to allow the synthesis tool to optimize an entire critical path in a single operation. Boolean optimization does not operate across hierarchical boundaries. Therefore, if a critical path is partitioned across boundaries, logic optimization is restricted. In addition, constraining modules is difficult if combinatorial logic is not restricted to the same level of hierarchy.

### Separate Speed Critical Paths from Non-critical Paths

To achieve satisfactory synthesis results, locate design modules with different functions at different levels of the hierarchy. Design speed is the first priority of optimization algorithms. To achieve a design that efficiently utilizes device area, remove timing constraints from design modules.

## Restrict Combinatorial Logic that Drives a Register to Same Hierarchy Level

To reduce the number of CLBs used, restrict combinatorial logic that drives a register to the same hierarchical block.

## Restrict Module Size

Restrict module size to 100 - 200 CLBs. This range varies based on your computer configuration; the time required to complete each optimization run; if the design is worked on by a design team; and the target FPGA routing resources. Although smaller blocks give you more control, you may not always obtain the most efficient design. For the final compilation of your design, you may want to compile fully from the top down. Check with your synthesis vendor for guidelines.

## Register All Outputs

Arrange your design hierarchy so that registers drive the module output in each hierarchical block. Registering outputs makes your design easier to constrain because you only need to constrain the clock period and the ClockToSetup of the previous module. If you have multiple combinatorial blocks at different levels of the hierarchy, you must manually calculate the delay for each module. Also, registering the outputs of your design hierarchy can eliminate any possible problems with logic optimization across hierarchical boundaries.

## Restrict One Clock to Each Module or to Entire Design

By restricting one clock to each module, you only need to describe the relationship between the clock at the top level of the design hierarchy and each module clock. By restricting one clock to the entire design, you only need to describe the clock at the top level of the design hierarchy.

***Note:*** See your synthesis tool documentation for more information on optimizing logic across hierarchical boundaries and compiling hierarchical designs.

Chapter 5

# Virtex-II Pro™ Considerations

This chapter includes coding techniques to help you improve synthesis results. It includes the following sections.

## Introduction

This chapter describes the special considerations that should be taken when simulating designs for Virtex-II Pro™ and Virtex-II Pro X™ FPGAs. The Virtex-II Pro™ family is a platform FPGA for designs that are based on IP cores and customized modules. The family incorporates RocketIO™ and PowerPC® CPU cores in Virtex-II Pro™ Series FPGA architecture.

The programable logic portion of the Virtex-II Pro™ family is based on Virtex-II™. While it is not bitstream or pin compatible, it can be programmed using the same methods as Virtex-II™ and Virtex-II™ designs, and can be implemented into Virtex-II Pro™ devices. In general, for details specific to designing for Virtex-II Pro™ and Virtex-II Pro X™, see the *Virtex-II Pro Platform FPGA User Guide* and the *RocketIO User Guide.*

## Using SmartModels to Simulate Virtex-II Pro™ Designs

SmartModels are an encrypted version to the actual HDL code. These models allow the user to simulate the actual functionality without having access to the code itself. The Xilinx® Virtex-II Pro™ family of devices gives the designer many new features, such as IBM® 's PowerPC® microprocessor and the RocketIO™. However, simulation of these new features requires the use of Synopsys® SmartModels along with the user design. This section gives the Virtex-II Pro™ simulation flow. It is assumed that the reader is familiar with the Xilinx® FPGA simulation flow.

## Simulation Components

The Virtex-II Pro™ device consists of several components. Each component has its own simulation model, and the individual simulation models must be correctly interconnected to get the simulation to work as expected. Following are the components that need to be simulated:

- FPGA Logic: This consists of either the RTL design constructed by the designer, or the back-annotated structural design created by the Xilinx® implementation tools.
- IBM® PowerPC® microprocessor: The microprocessor is simulated using SWIFT interface.
- RocketIO™: This is simulated using SWIFT interface.

## Overview of Virtex-II Pro™ Simulation Flow

The HDL simulation flow using Synopsys® SmartModels consists of three steps:

1. Instantiate the PowerPC® and / or RocketIO™ wrapper used for simulation and synthesis. During synthesis, the PowerPC® and the transceiver are treated as "black box" components. This requires that a wrapper be used that describes the modules port.
2. Install the SmartModels, if needed. See "Installing SmartModels from Xilinx® Implementation Tools" for details on installing SmartModels.
3. Use the SmartModels along with your design in an HDL simulator that supports the SWIFT interface.

You can find the instantiation wrapper files for the PowerPC® and RocketIO™ in the *Virtex-II Pro Platform FPGA Developers Kit* and the *RocketIO User Guide*.

The flow is shown in the following figure.

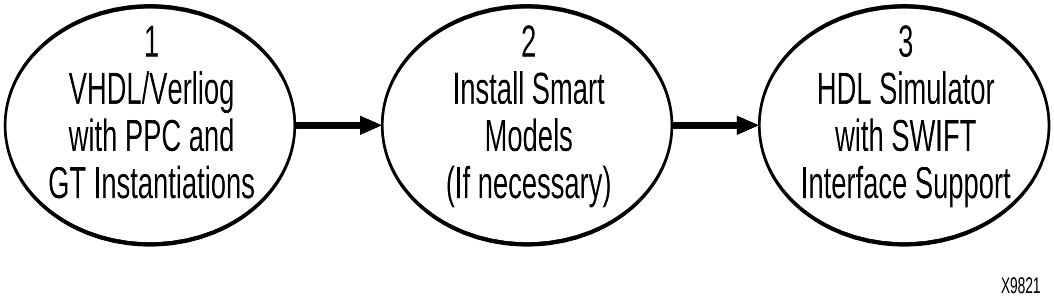


*Figure 5-1:* **HDL Simulation Flow for Virtex-II Pro™ Devices**

# SmartModels

The Xilinx® Virtex-II Pro™ simulation flow uses SmartModels for simulating the IBM® PowerPC® microprocessor and RocketIO™. SmartModels are simulator-independent models that are derived from the actual design and are therefore accurate evaluation models. To simulate these models, you must us a simulator that supports the SWIFT interface.

Synopsys® Logic Modeling uses the SWIFT interface to deliver models. SWIFT is a simulator- and platform-independent API developed by Synopsys® and adopted by all major simulator vendors, including Synopsys®, Cadence®, Mentor Graphics®, Model Technology® and others as a way of linking simulation models to design tools.

When running a back-annotated simulation, the precompiled SmartModels support gate-level, pin-to-pin, and back-annotation timing. Gate-level timing distributes the delays throughout the design, and all internal paths are accurately distributed. Multiple timing versions can be provided for different speed parts. Pin-to-pin timing is less accurate, but is faster since only a few top-level delays must be processed. Back-annotation timing allows the model to accurately process the interconnect delays between the model and the rest of the design. It can be used with either gate-level or pin-to-pin timing, or by itself.

You can find more details about SmartModels and the SWIFT interface in "Design Flow" volume of the *Virtex-II Pro Platform FPGA Developers Kit*, and on the Synopsys website at http://www.synopsys.com/products/designware/docs/.

# Supported Simulators

A simulator with SmartModel capability is required to use the SmartModels. While any HDL simulator that supports the Synopsys® SWIFT interface should be able to handle the Virtex-II Pro™ simulation flow, the following HDL simulators are officially supported by Xilinx® for Virtex-II Pro™ simulation.

## Solaris®

- MTI® ModelSim™ SE (5.5 and newer)
- MTI® ModelSim™ PE (5.7 and newer)
- Cadence® NC-Verilog™
- Cadence® NC-VHDL™
- Cadence® Verilog-XL™
- Synopsys® VCS™
- Synopsys® Scirocco™

## Windows®

- MTI® ModelSim™ SE (5.5)
- MTI® ModelSim™ PE (5.7 and newer)

# Required Software

To set up the simulation, install the Xilinx® implementation tools along with the simulator you are using.

## Solaris®

- Xilinx® Implementation Tools
- HDL Simulator that can simulate both VHDL/Verilog and SWIFT interface.

## Windows®

- IBM® CoreConnect™ Software. Details are available at http://www.xilinx.com/ipcenter/processor_central/register_coreconnect.htm
- HDL Simulator that can simulate both VHDL/Verilog and SWIFT interface.

# Installing SmartModels from Xilinx® Implementation Tools

The SmartModels are installed with the Xilinx® implementation tools, but they are not immediately ready for use. There are two ways to use them. In method one, use the precompiled models. The only additional setup necessary for this method is to set the LMC_HOME environment variable. Use this method if your design does not use any other vendors' Smart Models. In method two, install the PPC405 and GT SmartModel with additional SmartModels incorporated in the design. Compile all SmartModels into a common library for the simulator to use.

## Method One

### Windows®

For Windows®, the SmartModels are precompiled in the following directory.

```
Xilinx\smartmodel\nt\installed_nt
```

To use the precompiled SmartModels, set the following variable:

```
LMC_HOME = C:\Xilinx\smartmodel\nt\installed_nt
```

### Solaris®

For Solaris®, the SmartModels are precompiled in the following directory.

```
Xilinx/smartmodel/sol/installed_sol
```

To use the precompiled SmartModels set the following variables:

```
setenv LMC_HOME $XILINX/smartmodel/sol/installed_sol
setenv LMC_CONFIG $LMC_HOME/data/solaris.lmc
setenv LD_LIBRARY_PATH $LMC_HOME/lib/sun4Solaris.lib:$LD_LIBRARY_PATH
```

## Method Two

### Solaris®

1. Begin SmartModel Installation

   Run the `sl_admin.csh` program from the `$XILINX/verilog/smartmodel/sol/image` directory using the following commands:

   ```
   $ cd $XILINX/verilog/smartmodel/sol/image
   $ sl_admin.csh
   ```

2. Select SmartModels To Install

   a. The sl_admin GUI and the Set Library Directory dialog box appears. Change the default directory from `image/pcnt` to `installed`. Click **OK**. If the directory does not exist, the program asks if you want to create it. Click **OK**.

   b. The sl_admin GUI and the Install From... dialog box appears. Click **Open** to use the default directory.

   c. The Select Models to Install dialog box appears. Click **Add All** to select all models. Click **Continue**.

   d. The Select Platforms for Installation dialog box appears. For Platforms, select **Sun-4**. For EDAV Packages, select **Other**. Click **Install**.

   e. When the words “Install complete" appear, and the status line (bottom line of the sl_admin GUI) changes to Ready, installation is complete.

At this point the SmartModels have been installed. Exit the GUI by selecting **File->Exit** from the pull down menu, or use the GUI to perform other operations such as accessing documentation and running checks on your newly installed library.

To properly use the newly compiled models, set the LMC_HOME variable to the image directory. For example:

```
Setenv LMC_HOME $XILINX/verilog/smartmodel/sol/installed
```

### Windows®

1. Begin SmartModel Installation by running the `sl_admin.exe` program from the verilog`\smartmodel\nt\image\`*`pcnt`* directory.

2. Select SmartModels To Install.

   a. The sl_admin GUI and the Set Library Directory dialog box appears. Change the default directory from `image\pcnt` to `installed`. Click **OK**. If the directory does not exist, the program asks if you want to create it. Click **OK**.

   b. Click **Install** on the left side of the sl_admin window. This allows you choose the models to install.

   c. When the Install From... dialog box appears, click **Browse**, and select `sim_models\xilinx\verilog\smartmodel\nt\image` directory. Click **OK** to select that directory.

   d. The Select Models to Install dialog box appears. Click **Add All**, then click **OK.**

   e. The Choose Platform window appears. For Platforms, select **Wintel**. For EDAV Packages, select **Other**. Click **OK** to install.

   f. From the sl_admin window, you will see "Loading: gt_swift", and "Loading: ppc405_swift". When the words "Install complete" appear, installation is complete.

At this point, the SmartModels have been installed. Exit the GUI by selecting **File→ Exit** from the drop down menu, or use the GUI to perform other operations such as bringing up documentation and running checks on your newly installed library.

To properly use the newly compiled models, set the LMC_HOME variable to the image directory. For example:

```
Set LMC_HOME=$Xilinx$\verilog\smartmodel\nt\installed
```

For details specific to Virtex-II Pro™, see the *Virtex-II Pro Platform FPGA User Guide*.

# Running Simulation

This section describes how to set up and run simulation on the various supported simulators.

## MTI® ModelSim™ SE - Solaris®

### Simulator Setup

Although ModelSim™ SE supports the SWIFT interface, some modifications must be made to the default ModelSim™ setup to enable this feature. The ModelSim™ install directory contains an initialization file called `modelsim.ini`. In this initialization file, users can edit GUI and Simulator settings to default to their preferences. Parts of this `modelsim.ini` file must be edited to work properly along with the Virtex-II Pro™ device simulation models.

The following changes are needed in the `modelsim.ini` file. Make these changes to the `modelsim.ini` file located in the `$MODEL_TECH` directory. An alternative to making these edits is to change the MODELSIM environment variable setting in the MTI® setup script to point to the `modelsim.ini` file located in the each example's design directory.

1. After the lines:

   ```
   ; Simulator resolution
   ; Set to fs, ps, ns, us, ms, or sec with optional prefix of 1, 10, or
   100.
   ```

   Edit the statement that follows from Resolution = ns to Resolution = ps

2. After the lines:

   ```
   ; Specify whether paths in simulator commands should be described
   ; in VHDL or Verilog format. For VHDL, PathSeparator = /
   ; for Verilog, PathSeparator = .
   ```

   Comment the following statement called PathSeparator = / by adding a ";" at the start of the line.

3. After the line

   ```
   ; List of dynamically loaded objects for Verilog PLI applications add
   the following statement:
   Veriuser = $MODEL_TECH/libswiftpli.sl ;;  $DENALI/mtipli.so
   ```

4. After the line

   ```
   ;  Logic Modeling’s SmartModel SWIFT software (Sun4 Solaris 2.x)add the
   following statements:
   libsm = $MODEL_TECH/libsm.sl
   libswift = $LMC_HOME/lib/sun4Solaris.lib/libswift.so
   ```

***Note:*** It is important to make the changes in the order in which the commands appear in the `modelsim.ini file`. The simulation may not work if the order recommended above is not followed.

After editing the `modelsim.ini` file, add the following environment variable to the MTI® ModelSim™ SE setup script:

```
setenv MODELSIM /path_to_modelsim.ini_script/modelsim.ini
```

If the MODELSIM environment variable is not set properly, MTI® might not use this INI file, and then the initialization settings required for simulation will not be read by the simulator. Set up the MTI ModelSim™ SE simulation environment by sourcing the MTI SE setup script from the terminal.

### Running Simulation

In the `$xilinx/verilog/smartmodel/sol/simulation/mtiverilog` directory there are several files to help set up and run a simulation utilizing the SWIFT interface.

- `modelsim.ini` — An example `modelsim.ini` file used to set up ModelSim™ for SWIFT interface support. This file contains the changes outlined above. Xilinx® suggests that you make the changes to the `modelsim.ini` file located in the `$MODEL_TECH` directory, because of the library mappings included in this file.
- Setup — A script used to set the user environment for simulation and implementation. Here is an example of the variables set:

  ```
  setenv XILINX Xilinx_path
  setenv MODEL_TECH MTI_path
  setenv LM_LICENSE_FILE modelsim_license.dat;$LM_LICENSE_FILE
  setenv LMC_HOME ${XILINX}/verilog/smartmodel/sol/image
  setenv PATH
  ${LMC_HOME}/bin:${LMC_HOME}/lib/pcnt.lib:${MODEL_TECH}/bin:${XILINX}/b
  in/sol:{PATH}
  ```

  ***Note:*** The user is responsible for changing the parameters in *italics* to match the systems' configuration.
- Simulate — An example ModelSim™ simulation script. Illustrates which files need to be compiled and loaded for simulation. This file can be modified to simulate a design by including the design and test bench files appropriately. If the user's `modelsim.ini` file is being used, which contains the system mappings, the vmap commands can be commented out or deleted from this file.
- `run.do` — A script file used by the simulate script to run the complete simulation.

Once each of these files has been properly updated, you can run the simulation by sourcing the setup and simulate files.

## MTI® ModelSim™ SE - Windows®

### Simulator Setup

Although ModelSim™ SE supports the SWIFT interface, some modifications must be made to the default ModelSim™ setup to enable this feature. The ModelSim™ install directory contains an initialization file called `modelsim.ini`. In this initialization file, users can edit GUI and Simulator settings to default to their preferences. Parts of this `modelsim.ini` file must be edited to work properly with the Virtex-II Pro™ device simulation models.

The following changes are needed in the `modelsim.ini` file. These changes can be made to the `modelsim.ini` file located in the MODEL_TECH directory. An alternative to making these edits is to change the MODELSIM environment variable setting in the MTI® setup script to point to the `modelsim.ini` file located in the each example's design directory.

1. After the lines:

   ```
   ; Simulator resolution
   ; Set to fs, ps, ns, us, ms, or sec with optional prefix of 1, 10, or
   100.
   ```

   Change the Statement that follows from:

   ```
   Resolution = ns
   ```

   to:

   ```
   Resolution = ps
   ```

2. After the lines:

   ```
   ; Specify whether paths in simulator commands should be described
   ; in VHDL or Verilog format. For VHDL, PathSeparator = /
   ; for Verilog, PathSeparator = .
   ```

   Comment out the following statement:

   ```
   PathSeparator = /
   ```

   by adding a ";" at the start of the line.

3. After the line:

   ```
   ; List of dynamically loaded objects for Verilog PLI applications
   ```

   Add the following statement:

   ```
   Veriuser = %MODEL_TECH%/libswiftpli.dll
   ```

4. After the line:

   ```
   ; Logic Modeling’s SmartModel SWIFT software (Windows NT)
   ```

   add the following statements:

   ```
   libsm = %MODEL_TECH%/libsm.dll
   libswift = %LMC_HOME%/lib/pcnt.lib/libswift.dll
   ```

***Note:*** It is important to make these changes in the order in which the commands appear in the `modelsim.ini` file. The simulation may not work if the order recommended is not followed.

After editing the `modelsim.ini` file, add the following environment variable to the MTI® ModelSim™ SE setup script:

```
set MODELSIM=path_to_modelsim.ini_script\modelsim.ini
```

If the MODELSIM environment variable is not set properly, MTI® might not use this INI file, and then the initialization settings required for simulation will not be read by the simulator. Set up the MTI® ModelSim™ SE simulation environment by sourcing the MTI SE setup script from the terminal.

## Running Simulation

In the `$XILINX\verilog\smartmodel\sol\simulation\mtiverilog` directory there are several files to help you set up and run a simulation using the SWIFT interface.

- `modelsim.ini` — An example `modelsim.ini` file used to set up ModelSim™ for SWIFT interface support. This file contains the changes outlined above. We suggest that you make the changes to the `modelsim.ini` file located in the `$MODEL_TECH` directory, because of the library mappings included in this file.

- Setup — Description of variables, which a user must set for correct simulation and implementation. Here is an example of the variables set:

  ```
  set XILINX Xilinx_path
  set LMC_HOME %XILINX%\verilog\smartmodel\sol\image
  set MODEL_TECH MTI_path
  set LM_LICENSE_FILE license.dat;%LM_LICENSE_FILE%
  set path %LMC_HOME%\bin;%LMC_HOME%\lib\pcnt.lib;%MODEL_TECH%
  \bin;%XILINX%\bin\nt;%path%
  ```

  ***Note:*** The user is responsible for changing the parameters in *italics* to match the systems' configuration.

- `simulate.bat` — An example ModelSim™ simulation script. Illustrates which files must be compiled and loaded for simulation. This file can be modified to simulate a design by including the design and test bench files appropriately. If the user's `modelsim.ini` file is being used, which contains the system mappings, the vmap commands can be commented out or deleted from this file.
- `run.do` — A script file used by the simulate script to run the complete simulation.

Once each of these files has been properly updated, run the simulation by double-clicking `simulate.bat`.

## Cadence® Verilog-XL™ - Solaris®

### Running Simulation

A Verilog-XL simulation incorporating the SWIFT interface canbe initiated in two ways.

1. In the `$XILINX/verilog/smartmodel/sol/simulation/verilogxl` directory there are several files to help set up and run a simulation using the SWIFT interface.

   - Setup — Description of variables, which a user must set for correct simulation and implementation. Here is an example of the variables set:

   ```
   setenv XILINX Xilinx_path
   setenv LM_LICENSE_FILE verilogxl_license.dat:${LM_LICENSE_FILE}
   setenv CDS_INST_DIR Cadence_path
   setenv LD_LIBRARY_PATH
   ${V2PRO}/source/sim_models/Xilinx/verilog/smartmodel/sol/installed/lib
   /sun4Solaris.lib:${LD_LIBRARY_PATH}
   setenv LMC_CDS_VCONFIG ${CDS_INST_DIR}/tools.sun4v/verilog/bin/vconfig
   setenv LM_LICENSE_FILE license.dat:${LM_LICENSE_FILE}
   setenv PATH ${LMC_HOME}/bin ${CDS_INST_DIR}/tools/bin $PATH
   setenv PATH ${XILINX}/bin/sol ${PATH}
   ```

     ***Note:*** The user is responsible for changing the parameters in *italics* to match the systems configuration. The `LD_LIBRARY_PATH` variable must point to the SmartModel installation directory.

   - Simulate — An example Verilog-XL compilation simulation script. Illustrates which files need to be compiled and loaded for simulation. This file can be modified to simulate a design by including the design and test bench files appropriately. The user should add `+loadpli1=swiftpli:swift_boot`, a Verilog directive, to the simulate script. For example:

     ```
     verilog +loadpli1=swiftpli:swift_boot \
     ```

   Once each of these files has been properly updated, you can run the simulation.

2. This flow requires administrative privileges and is not recommended.

   In the `$XILINX/verilog/smartmodel/sol/simulation/verilogxl` directory there are several files to help set up and run a simulation using the SmartModels. A description of each file follows.

   - Readme — Outlines the steps of the secondary flow to utilize the SWIFT interface.

   a. Edit the setup file, as shown below, to set up environment for Verilog-XL.

   ```
   source setup
   ```

   ***Note:*** The following step is not required if the models have been installed.

   b.

   ```
   cd $XILINX/verilog/smartmodel/sol/image
   ```

   ```
   Enter: sl_admin.csh
   ```

   c.

   ```
   Enter: pliwiz
   ```

   ```
   Config Session Name - xilinx
   ```

   ```
   Verilog-XL
   ```

   ```
   Stand Alone
   ```

   ```
   SWIFT Interface
   ```

   ```
   Finish
   ```

   ```
   No
   ```

   d.

   ```
   cp -p $CDS_INST_DIR/tools/pliwizard/src/Makefile.xl.sun4v .
   ```

   e.

   ```
   edit Makefile_pliwiz.xl
   ```

   f.

   ```
   change $(INSTALL_DIR)/tools/pliwizard/src/Makefile.xl.sun4v to
   ./Makefile.xl.sun4v
   ```

   g.

   ```
   edit Makefile.xl.sun4v
   ```

   ```
   Change CC = cc to CC = gcc
   ```

   h.

   ```
   make all
   ```

   i. edit the simulate file

   ```
   source simulate
   ```

   - Setup — Description of variables, which a user must set for correct simulation and implementation. Here is an example of the variables set:

```
setenv XILINX Xilinx_path
setenv LM_LICENSE_FILE verilogxl_license.dat:${LM_LICENSE_FILE}
setenv CDS_INST_DIR Cadence_path

setenv LD_LIBRARY_PATH
${V2PRO}/source/sim_models/Xilinx/verilog/smartmodel/sol/installed/lib
/sun4Solaris.lib:${LD_LIBRARY_PATH}

setenv LMC_CDS_VCONFIG ${CDS_INST_DIR}/tools.sun4v/verilog/bin/vconfig
setenv LM_LICENSE_FILE license.dat:${LM_LICENSE_FILE}

setenv PATH ${LMC_HOME}/bin ${CDS_INST_DIR}/tools/bin $PATH
setenv PATH ${XILINX}/bin/sol ${PATH}
```

***Note:*** The user is responsible for changing the parameters in *italics* to match the systems' configuration. The LD_LIBRARY_PATH variable must be pointing to the SmartModel installation directory.

- Simulate— An example Verilog-XL compilation simulation script. Illustrates which files need to be compiled and loaded for simulation. This file can be modified to simulate a design by including the design and test bench files appropriately.

## Cadence® NC-Verilog™ - Solaris®

### Running Simulation

In the `$XILINX/verilog/smartmodel/sol/simulation/ncverilog` directory there are several files to help set up and run a simulation utilizing the SWIFT interface.

- Setup — Description of variables that a user must set for correct simulation and implementation. Here is an example of the variables set:

```
setenv XILINX Xilinx_path
setenv CDS_INST_DIR Cadence_path
setenv LM_LICENSE_FILE license.dat:$LM_LICENSE_FILE

setenv LMC_HOME $XILINX/verilog/smartmodel/sol/image
setenv LMC_CONFIG $LMC_HOME/data/solaris.lmc

setenv LD_LIBRARY_PATH $CDS_INST_DIR/tools.sun4v/lib:$LD_LIBRARY_PATH
setenv LMC_CDS_VCONFIG $CDS_INST_DIR/tools.sun4v/verilog/bin/vconfig

setenv PATH ${LMC_HOME}/bin ${CDS_INST_DIR}/tools/bin ${PATH}
setenv PATH ${XILINX}/bin/sol ${PATH}
```

***Note:*** The user is responsible for changing the parameters in *italics* to match the systems' configuration.

- Simulate — An example NC-Verilog™ compilation simulation script. Illustrates which files need to be compiled and loaded for simulation. This file can be modified to simulate a design by including the design and test bench files appropriately.

  Once each of these files has been properly updated, run the simulation.

## Synopsys® VCS™ - Solaris®

### Running Simulation

In the `$XILINX/verilog/smartmodel/sol/simulation/vcs` directory there are several files to help set up and run a simulation utilizing the SWIFT interface.

- Setup — Description of variables, which a user must set for correct simulation and implementation. Here is an example of the variables set:

  ```
  setenv XILINX Xilinx_path
  setenv VCS_HOME VCS_path
  setenv LM_LICENSE_FILE license.dat:${LM_LICENSE_FILE}
  setenv LMC_HOME ${XILINX}/verilog/smartmodel/sol/image
  setenv LMC_CONFIG ${LMC_HOME}/data/solaris.lmc
  setenv VCS_CC gcc
  setenv PATH ${LMC_HOME}/bin ${VCS_HOME}/bin ${PATH}
  setenv PATH ${XILINX}/bin/sol ${PATH}
  ```

  ***Note:*** The user is responsible for changing the parameters in *italics* to match the systems' configuration.

- Simulate — Example Verilog-XL compilation simulation script. Illustrates which files need to be compiled and loaded for simulation. This file can be modified to simulate a design by including the design and test bench files appropriately.

  Once each of these files has been properly updated, run the simulation.

Chapter 6

# Verifying Your Design

This chapter describes the basic HDL simulation flow using the Xilinx® and third party software. It includes the following sections.

## Introduction

Increasing design size and complexity, as well as recent improvements in design synthesis and simulation tools, have made HDL the preferred design language of most integrated circuit designers. The two leading HDL synthesis and simulation languages today are Verilog and VHDL. Both of these languages have been adopted as IEEE standards.

The Xilinx® implementation tools software is designed to be used with several HDL synthesis and simulation tools that provide a solution for programmable logic designs from beginning to end. The Xilinx® software provides libraries, netlist readers and netlist writers along with the powerful place and route software that integrates with your HDL design environment on PC and UNIX workstation platforms.

# Adhering to Industry Standards

The standards in the following table are supported by the Xilinx® simulation flow.

*Table 6-1:* **Standards Supported by Xilinx® Simulation Flow**

| Description | Version |
|---|---|
| VHDL Language | IEEE-STD-1076-1993 |
| VITAL Modeling Standard | IEEE-STD-1076.4-2000 |
| Verilog Language | IEEE-STD-1364-2001 |
| Standard Delay Format (SDF) | OVI 3.0 |
| Std_logic Data Type | IEEE-STD-1164-93 |

The Xilinx® software currently supports the Verilog IEEE 1364 2001 Standard, VHDL IEEE Standard 1076-1993 and IEEE Standard 1076.4-2000 for Vital (Vital 2000), and SDF version 3.0.

***Note:*** Although the Xilinx® HDL netlisters produce IEEE-STD-1076-93 VHDL code or IEEE-STD-1364-2001 Verilog code, that does not restrict the use of newer or older standards for the creation of test benches or other simulation files. If the simulator being used supports both older and newer standards, then generally, both standards can be used in these simulation files. Be sure to indicate to the simulator during code compilation which standard was used for the creation of the file.

Xilinx® currently tests and supports the following simulators for VHDL and Verilog simulation:

- VHDL
  - MTI® ModelSim™
  - Cadence® NC-VHDL™
  - Synopsys® Scirocco™
- Verilog
  - MTI® ModelSim™
  - Cadence® Verilog-XL™
  - Cadence® NC-Verilog™
  - Synopsys® VCSi™

In general, you should run the most current version of the simulator available to you.

Xilinx® develops its libraries and simulation netlists using IEEE standards, so you should be able to use most modern VHDL and Verilog simulators. Check with your simulator vendor before you start to confirm that the proper standards are supported by your simulator, and to verify the proper settings for your simulator.

The Xilinx® VHDL libraries are tied to the IEEE-STD-1076.4-2000 VITAL standard for simulation acceleration. This VITAL 2000 is in turn based on the IEEE-STD-1076-93 VHDL language. Because of this the Xilinx® libraries must be compiled as 1076-93.

VITAL libraries include some additional processing for timing checks and back-annotation styles. The UNISIM library turns these timing checks off for unit delay functional simulation. The SIMPRIM back-annotation library keeps these checks on by default to allow accurate timing simulations.

# Simulation Points

Xilinx® supports functional and timing simulation of HDL designs at five points in the HDL design flow. Figure 6-1 shows the points of the design flow. All five points are described in the following section.

1. Register Transfer Level (RTL) simulation, which may include the following:
   - RTL Code
   - Instantiated UNISIM library components
   - XilinxCoreLib and UNISIM gate-level models (CORE Generator™)
   - SmartModels
2. Post-synthesis functional simulation, which may include one of the following (optional):
   - Gate-level netlist containing UNISIM library components (written by the synthesis tool)
   - XilinxCoreLib and UNISIM gate-level models (CORE Generator™)
   - SmartModels
3. Post-NGDBUILD Simulation (optional):
   - Gate-level netlist containing SIMPRIM library components
   - SmartModels
4. Post-Map with partial back-annotated timing without routing delays, which may include the following (optional):
   - Gate-level netlist containing SIMPRIM library components
   - Standard Delay Format (SDF) files
   - SmartModels
5. Post-Place and Route with full back-annotated timing, which may include the following:
   - Gate-level netlist containing SIMPRIM library components
   - Standard Delay Format (SDF) files
   - SmartModels

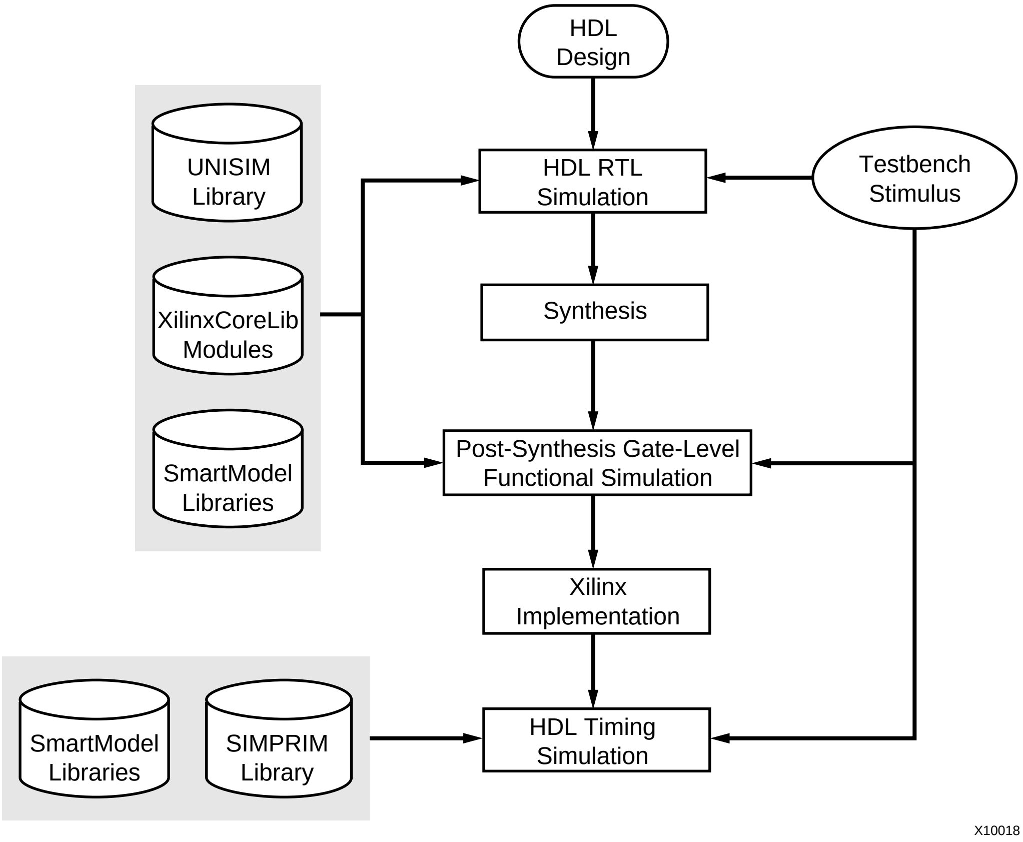


*Figure 6-1:* **Primary Simulation Points for HDL Designs**

The Post-NGDBuild and Post-Map simulations can be used when the synthesis tool either cannot write VHDL or Verilog, or if the netlist is not in terms of UNISIM components.

*Table 6-2:* **Five Simulation Points in HDL Design Flow**

| Simulation | | UNISIM | XilinxCoreLib Models | SmartModel | SIMPRIM | SDF |
|---|---|---|---|---|---|---|
| 1. | RTL | X | X | X | | |
| 2. | Post-Synthesis (optional) | X | X | X | | |
| 3. | Functional Post-NGDBuild (optional) | | | X | X | |
| 4. | Functional Post-Map (optional) | | | X | X | X |
| 5. | Post-Route Timing | | | X | X | X |

These Xilinx® simulation points are described in detail in the following sections. The libraries required to support the simulation flows are described in detail in "VHDL / Verilog Libraries and Models". The flows and libraries support functional equivalence of initialization behavior between functional and timing simulations.

Different simulation libraries are used to support simulation before and after running NGDBuild. Prior to NGDBuild, your design is expressed as a UNISIM netlist containing Unified Library components that represents the logical view of the design. After NGDBuild, your design is a netlist containing SIMPRIMs that represents the physical view of the design. Although these library changes are fairly transparent, there are two important considerations to keep in mind: first, you must specify different simulation libraries for pre- and post-implementation simulation, and second, there are different gate-level cells in pre- and post-implementation netlists.

For Verilog, within the simulation netlist there is the Verilog system task `$sdf_annotate`, which specifies the name of the SDF file to be read. If the simulator supports the `$sdf_annotate` system task, the Standard Delay Format (SDF) file is automatically read when the simulator compiles the Verilog simulation netlist. If the simulator does not support `$sdf_annotate`, in order to get timing values applied to the gate-level netlist, you must manually specify to the simulator to annotate the SDF file. Consult your simulation documentation to find the proper method to annotate SDF files.

For VHDL, you must specify the location of the SDF file and which instance to annotate during the timing simulation. The method for doing this is different depending on the simulator being used. Typically, a command line or GUI switch is used to read the SDF file. Consult your simulation documentation to find the proper method to annotate SDF files.

## Register Transfer Level (RTL)

The RTL-level (behavioral) simulationenables you to verify or simulate a description at the system or chip level. This first pass simulation is typically performed to verify code syntax and to confirm that the code is functioning as intended. At this step, no timing information is provided and simulation should be performed in unit-delay mode to avoid the possibility of a race condition.

RTL simulation is not architecture-specific unless the design contains instantiated UNISIM or CORE Generator™ components. To support these instantiations, Xilinx® provides the UNISIM and XilinxCoreLib libraries. You can instantiate CORE Generator™ components if you do not want to rely on the module generation capabilities of the synthesis tool, or if the design requires larger memory structures.

A general suggestion for the initial design creation is to keep the code behavioral. Avoid instantiating specific components unless necessary. This allows for more readable code, faster and simpler simulation, code portability (the ability to migrate to different device families), and code reuse (the ability to use the same code in future designs). However, you may find it necessary to instantiate components if the component is not inferrable (i.e. DCM, GT, PPC405, etc.), or in order to control the mapping, placement or structure of a function.

## Post-Synthesis (Pre-NGDBuild) Gate-Level Simulation

Most synthesis tools have the ability to write out a post-synthesis HDL netlist for a design. If the VHDL or Verilog netlists are written for UNISIM library components, you may then use the netlists to simulate the design and evaluate the synthesis results. However, Xilinx® does not support this method if the netlists are written in terms of the vendor's own simulation models.

The instantiated CORE Generator™ models are used for any post-synthesis simulation because these modules are processed as a "black box" during synthesis. It is important that you maintain the consistency of the initialization behavior with the behavioral model used for RTL, post-synthesis simulation, and the structural model used after implementation. In addition, the initialization behavior must work with the method used for synthesized logic and cores.

## Post-NGDBuild (Pre-Map) Gate-Level Simulation

The post-NGDBuild (pre-map) gate-level functional simulation is used when it is not possible to simulate the direct output of the synthesis tool. This occurs when the tool cannot write UNISIM-compatible VHDL or Verilog netlists. In this case, the NGD file produced from NGDBUILD is the input into the Xilinx® simulation netlister, NetGen. NetGen creates a structural simulation netlist based on SIMPRIM models.

Like post-synthesis simulation, pre-NGDBuild simulation allows you to verify that your design has been synthesized correctly, and you can begin to identify any differences due to the lower level of abstraction. Unlike the post-synthesis pre-NGDBuild simulation, there are GSR and GTS nets that must be initialized, just as for post-Map and post-PAR simulation. Please refer to the "Understanding the Global Reset and Tristate for Simulation" section for details on using the GSR and GTS signals for post-NGDBuild simulation.

## Post-Map Partial Timing (CLB and IOB Block Delays)

You may also perform simulation after mapping the design. Post-Map simulation occurs before placing and routing. This simulation includes the block delays for the design, but not the routing delays. This is generally a good metric to test whether the design is meeting the timing requirements before additional time is spent running the design through a complete place and route.

As with the post-NGDBuild simulation, NetGen is used to create the structural simulation netlist based on SIMPRIM models.

When you run the simulation netlister tool, NetGen, an SDF file is created. The delays for the design are stored in the SDF file which contains all block or logic delays. However, it does not contain any of the routing delays for the design since the design has not yet been placed and routed. As with all Netgen created netlists, GSR and GTS signals must be accounted for. Please refer to the "Understanding the Global Reset and Tristate for Simulation" section for details on using the GSR and GTS signals for post-NGDBuild simulation.

## Timing Simulation Post-Place and Route Full Timing (Block and Net Delays)

After your design has completed the place and route process in the Xilinx® Implementation Tools, a timing simulation netlist can be created. It is not until this stage of design implementation that you start to see how your design behaves in the actual circuit. The overall functionality of the design was defined in the beginning stages, but it is not until the design has been placed and routed that all of the timing information of the design can be accurately calculated.

The previous simulations that used NetGen created a structural netlist based on SIMPRIM models. However, this netlist comes from the placed and routed NCD file. This netlist has GSR and GTS nets that must be initialized. For more information on initializing the GSR and GRTS nets, please refer to "Understanding the Global Reset and Tristate for Simulation" in this chapter.

When you run timing simulation, an SDF file is created as with the post-Map simulation. However, this SDF file contains all block and routing delays for the design.

## Providing Stimulus

Before you perform simulation, you should create a test bench or test fixture to apply the stimulus to the design. A test bench is HDL code written for the simulator that instantiates the design netlist(s), initializes the design and then applies stimuli to verify the functionality of the design. You can also set up the test bench to display the desired simulation output to a file, waveform or screen.

The test bench has many advantages over interactive simulation methods. For one, it allows repeatable simulation throughout the design process. It also provides documentation of the test conditions.

There are several methods to create a test bench and simulate a design. A test bench can be very simple in structure and sequentially apply stimulus to specific inputs. A test bench may also be very complex, including subroutine calls, stimulus read in from external files, conditional stimulus or other more complex structures.

The ISE tools create a template test bench containing the proper structure, library references, and design instantiation based on your design files from Project Navigator. This greatly eases test bench development at the beginning stages of the design.

Alternately, you may use the HDL Bencher tool in ISE to automatically create a test bench by drawing the intended stimulus and the expected outputs in a waveform viewer. Please refer to the *ISE* and / or *HDL Bencher Online Help* for more information.

With yet another method, you can use NetGen to create a test bench file. The –tb switch for Netgen creates a test fixture or test bench template. The Verilog test fixture file has a .tv extension, and the VHDL test bench file has a .tvhd extension.

Xilinx® recommends giving the name *testbench* to the main module or entity name in the test bench file. Xilinx® also suggests, but does not require, that you specify the instance name for the instantiated top-level of the design in the test bench as *UUT*. These names are consistent with the default names used by ISE for calling the test bench and annotating the SDF file when invoking the simulator.

Xilinx® recommends that you initialize all inputs to the design within the test bench at simulation time zero in order to properly start simulation with known values. Xilinx® also suggests that you not begin applying stimulus data until after 100 ns in order to account for the default Global Set / Reset pulse used in SIMPRIM-based simulation. However, the clock source should begin before the GSR is released. For more information on GSR, refer to the "Understanding the Global Reset and Tristate for Simulation" section.

## VHDL/Verilog Libraries and Models

The five simulation points listed previously require the UNISIM, CORE Generator™ (XilinxCoreLib), SmartModel and SIMPRIM libraries.

The first point, RTL simulation, is a behavioral description of your design at the register transfer level. RTL simulation is not architecture-specific unless your design contains instantiated UNISIM, or CORE Generator™ components. To support these instantiations, Xilinx® provides a functional UNISIM library, a CORE Generator™ Behavioral XilinxCoreLib library and a SmartModelLibrary™. You can also instantiate CORE Generator™ components if you do not want to rely on the module generation capabilities of your synthesis tool, or if your design requires larger memory structures.

The second simulation point is post-synthesis (pre-NGDBuild) gate-level simulation. If the UNISIM library and CORE Generator™ components are used, then the UNISIM, the XilinxCorLib and SmartModelLibraries must all be used. The synthesis tool must write out the HDL netlist using UNISIM primitives. Otherwise, the synthesis vendor provides its own post-synthesis simulation library.

The third, fourth, and fifth points (post-NGDBuild, post-map, and post-route) use the SIMPRIM and the SmartModelLibraries. The following table indicates what library is required for each of the five simulation points.

*Table 6-3:* **Simulation Phase Library Information**

| Simulation Point | Compilation Order of Library Required |
|---|---|
| RTL | UNISIM<br>XilinxCoreLib<br>SmartModel |
| Post-Synthesis | UNISIM<br>XilinxCoreLib<br>SmartModel |
| Post-NGDBuild | SIMPRIM<br>SmartModel |
| Post-Map | SIMPRIM<br>SmartModel |
| Post-Route | SIMPRIM<br>SmartModel |

## Locating Library Source Files

The following table provides information on the location of the simulation library source files, as well as the order for a typical compilation.

*Table 6-4:* **Simulation Library Source Files**

| Library | Location of Source Files | | Compile Order | |
|---|---|---|---|---|
| | Verilog | VITAL VHDL | Verilog | VITAL VHDL |
| UNISIM<br>Spartan-II™, Spartan-IIE™, Spartan-3™, Virtex™, Virtex-E™, Virtex-II™, Virtex-II Pro™, Virtex-II Pro X™ | `$XILINX/verilog/src/unisims` | `$XILINX/vhdl/src/unisims` | No special compilation order required for Verilog libraries | Required;<br>typical compilation order:<br>`unisim_VCOMP.vhd`<br>`unisim_VPKG.vhd`<br>`unisim_VITAL.vhd` |
| UNISIM<br>9500™, CoolRunner™, CoolRunner-II™, CoolRunner-IIS™ | `$XILINX/verilog/src/uni9000` | `$XILINX/vhdl/src/unisims` | No special compilation order required for Verilog libraries | Required;<br>typical compilation order:<br>`unisim_VCOMP.vhd`<br>`unisim_VPKG.vhd`<br>`unisim_VITAL.vhd` |
| XilinxCoreLib<br>FPGA Families only | `$XILINX/verilog/src/XilinxCoreLib` | `$XILINX/vhdl/src/XilinxCoreLib` | No special compilation order required for Verilog libraries | Compilation order required;<br>See the vhdl_analyze_order file located in `$XILINX/vhdl/src/XilinxCoreLib/` for the required compile order |

*Table 6-4:* **Simulation Library Source Files**

| Library | Location of Source Files | | Compile Order | |
|---|---|---|---|---|
| | Verilog | VITAL VHDL | Verilog | VITAL VHDL |
| SmartModel<br>Virtex-II Pro™ Virtex-II Pro X™ | $XILINX/ smartmodel/ *platform/* wrappers/ *simulator* | $XILINX/ smartmodel/ *platform/* wrappers/ *simulator* | No special compilation order required for Verilog libraries | Required.<br>Typical compilation order for Functional Simulation:<br>unisim_VCOMP.vhd<br>smartmodel_wrappers.vhd<br>unisim_SMODEL.vhd<br>Typical compilation order for Timing Simulation:<br>simprim_Vcomponents.vhd<br>smartmodel_wrappers.vhd<br>simprim_SMODEL.vhd |
| SIMPRIM<br>(All Xilinx® Technologies) | $XILINX/verilog /src/simprims | $XILINX/vhdl/ src/simprims | No special compilation order required for Verilog libraries | Required;<br>typical compilation order:<br>simprim_Vcomponents.vhd<br>simprim_Vpackage.vhd<br>simprim_VITAL.vhd |

## Using the UNISIM Library

The UNISIM Library is used for functional simulation only. This library includes all of the Xilinx® Unified Library primitives that are inferred by most synthesis tools. In addition, the UNISIM Library includes primitives that are commonly instantiated, such as DCMs, BUFGs and GTs. You should generally infer most design functionality using behavioral RTL code unless the desired component is not inferrable by your synthesis tool, or you want to take manual control of mapping and/or placement of a function.

### UNISIM Library Structure

The UNISIM library structure is different for VHDL and Verilog. The VHDL UNISIM library is split into four files containing the component declarations (unisim_VCOMP.vhd), package files (unisim_VPKG.vhd), entity and architecture declarations (unisim_VITAL.vhd), and SmartModel declarations (unisim_SMODEL.vhd). All primitives for all Xilinx® device families are specified in these files. The VHDL UNISIM Library source files are located at $XILINX/vhdl/src/unisims.

For Verilog, each library component is specified in a separate file. The reason for this is to allow automatic library expansion using the `uselib compiler directive or the –y library specification switch. All Verilog module names and file names are all upper case (i.e. module BUFG would be BUFG.v, module IBUF would be IBUF.v). Since Verilog is a case-sensitive language, ensure that all UNISIM primitive instantiations adhere to this upper-case naming convention.

The library sources are split into two directories in which the FPGA device families (Spartan-II™, Spartan-IIE™, Spartan-3™, Virtex™, Virtex™- E, Virtex-II™, Virtex-II Pro™, Virtex-II Pro X™) are located at `$XILINX/verilog/src/unisims`, and the CPLD device families (XC9500XL™, XC9500XV™, CoolRunner-XPLA3™, CoolRunner-II™, CoolRunner-IIS™) are located at `$XILINX/verilog/src/uni9000`.

## Using the CORE Generator™ XilinxCoreLib Library

The Xilinx® CORE Generator™ is a graphical intellectual property design tool for creating high-level modules like FIR Filters, FIFOs and CAMs, as well as other advanced IP. You can customize and pre-optimize modules to take advantage of the inherent architectural features of Xilinx® FPGA devices, such as block multipliers, SRLs, fast carry logic, and on-chip, single-port or dual-port RAM. You can also select the appropriate HDL model type as output to integrate into your HDL design.

The CORE Generator™ HDL library models are used for RTL simulation. The models do not use library components for global signals.

### CORE Generator™ Library Structure

The VHDL CORE Generator™ library source files are found in `$XILINX/vhdl/src/XilinxCoreLib`.

The Verilog CORE Generator™ library source files are found in `$XILINX/verilog/src/XilinxCoreLib`.

## Using the SIMPRIM Library

The SIMPRIM library is used for post Ngdbuild (gate level functional), post-Map (partial timing), and post-place-and-route (full timing) simulations. This library is architecture independent.

### SIMPRIM Library Structure

The VHDL SIMPRIM Library source files are found in `$XILINX/vhdl/src/simprims`.

The Verilog SIMPRIM Library source files are found in `$XILINX/verilog/src/simprims`.

## Using the SmartModel Library™

The SmartModel Libraries™ are used to model very complex functions of modern FPGAs such as the PowerPC® (PPC) and the RocketIO™. SmartModels are encrypted source files that communicate with simulators via the SWIFT interface. The SmartModel Libraries™ are located at `$XILINX/smartmodel`, and require additional installation steps to properly install on your system. Additional setup within the simulator may also be required. Please refer to Chapter 5, “Virtex-II Pro™ Considerations” for more information on proper installation and setup of the SmartModel Libraries™.

# Compiling Xilinx® Simulation Libraries (COMPXLIB)

Before starting the functional simulation of your design, you must compile the Xilinx® Simulation Libraries for the target simulator. For this purpose Xilinx® provides a tool called COMPXLIB.

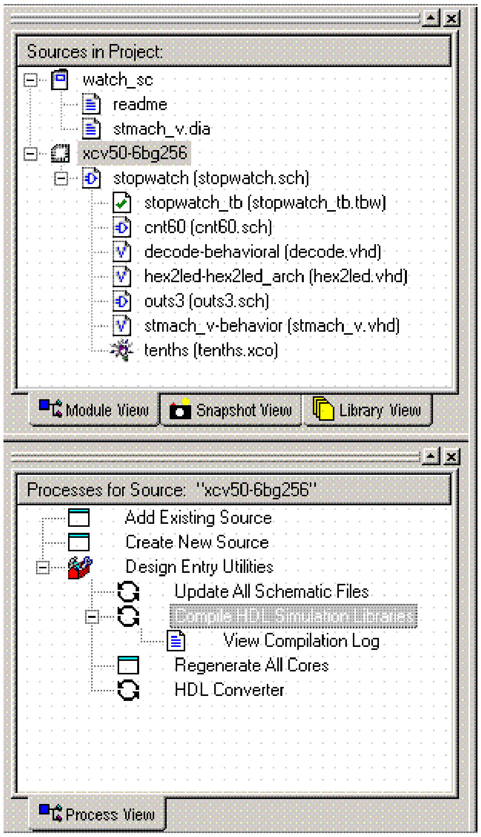
COMXPLIB is a tool for compiling the Xilinx® HDL based simulation libraries using the tools provided by the simulator vendor.

## Compiling Simulation Libraries

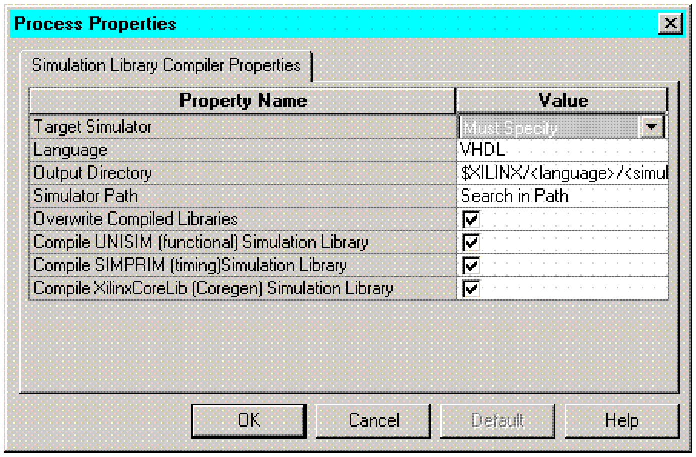
You can compile the libraries from Project Navigator or from the command line, as described in this section.

### From Project Navigator

1. Create or open an existing project for Project Navigator.
2. In the Sources in Project window, highlight the target device.
3. In the Processes for Source window, under the Design Entry Utilities toolbox, right-click **Compile HDL Simulation Libraries** and select **Properties** to open the Process Properties dialog box.



4. Choose one or more of the following options from the Process Properties dialog box:



| Property Name | Value |
|---|---|
| Target Simulator | Must Specify |
| Language | VHDL |
| Output Directory | $XILINX/<language>/<simul |
| Simulator Path | Search in Path |
| Overwrite Compiled Libraries | ☑ |
| Compile UNISIM (functional) Simulation Library | ☑ |
| Compile SIMPRIM (timing)Simulation Library | ☑ |
| Compile XilinxCoreLib (Coregen) Simulation Library | ☑ |

***Note:*** Project Navigator will only show the options that apply to your specific design flow. For example, if you have created a Virtex-II™ project, it will only show you the list of libraries that are required for simulating a Virtex-II™ project.

- Target Simulator

  The Target Simulator property allows you to select the target simulator for which the libraries are to be compiled. Click anywhere in the Value field to display the list of supported simulators.

  ***Note:*** You must choose a Target Simulator before starting the process.

- Language

  By default, the Language property is selected according to the Project Properties.

- Output Directory

  The Output Directory property allows you to specify the directory where the compiled libraries will be saved.

  To change the directory path, type a new path in the Value field, or click anywhere in the Value field and double-click the small button that appears to the right of the current value. Use the Browse for File dialog box that appears to choose an output directory.

  The default directory path is `$XILINX/`*`language`*`/`*`simulator`*, where *language* is the selected language to compile and *simulator* is the name of the selected simulator.

- Simulator Path

  The Simulator Path property allows you to specify the simulator installation bin path where the simulator executables reside.

  To change the simulator path, click anywhere in the Value field and double-click the small button that appears to the right of the current value. Use the Browse for File dialog box that appears to choose a directory.

  By default, the path to the simulator will be searched for via the path environment variable. Please note that if you have multiple simulator installation versions on your system, or if you do not have the simulator executable path defined in your environment variable, you can use this option to specify the executable path to your appropriate simulator.

- Overwrite Compiled Library

  The Overwrite Compiled Library property allows you to overwrite the previously compiled library. Select the check box in the Value field to enable overwriting. By default, the check box is checked.

  - Compile *library_name* Library

    This is a list of libraries that allows you to select or deselect the libraries that you want to compile.

    By default, all the libraries will be selected. Please note that the list of libraries depends upon the type of device family selected in the Project Properties.

5. Click **OK**. The Compile HDL Simulation Libraries properties are now set.
6. Double-click **Compile HDL Simulation Libraries**. Project Navigator will now compile the libraries using the properties specified in the properties you have set.

After the process is completed, double-click **View Compilation Log** to open the COMPXLIB.log file to view the compilation results.

### From Command Line

To compile libraries, type the following on the command line:

```
compxlib [options]
```

See "COMPXLIB Syntax" for options and syntax details.

To view COMPXLIB help, type the following on the command line:

```
compxlib -help
```

## Library Support

COMPXLIB supports the compilation of the following Xilinx® HDL Simulation Libraries:

- UNISIM (Functional)
- Uni9000 (Timing)
- SIMPRIM (Timing)
- XilinxCoreLib (Functional)
- SmartModel Library™(Functional & Timing)
- CoolRunner™ (Functional)
- Abel (Functional)

## Device Family Support

COMPXLIB supports the compilation of libraries for all Xilinx® Device Families.

## Simulator Support

COMPXLIB supports the compilation of Xilinx® HDL Simulation Libraries for the following simulators:

- ModelSim™ SE (all Xilinx® supported platforms)
- ModelSim™ PE (all Xilinx® supported platforms)
- NCSIM™ (all Xilinx® supported platforms)

- VCS™ (only on Solaris® and Linux based platforms)
- VCSi™ (only on Solaris® and Linux based platforms)
- Scirocco™ (only on Solaris® and Linux based platforms)

***Note:*** The VHDL SIMPRIM library in this release is VITAL2000 compliant. Ensure that your simulator is also VITAL2000 compliant to successfully compile the SIMPRIM library.

***Note:*** If using ModelSim™-XE, device library compilation is not necessary as the libraries are pre-compiled and installed with the software. For updates for the ModelSim™-XE library, please consult the Xilinx® support website at:http://support.xilinx.com.

## COMPXLIB Syntax

The following command compiles all Xilinx® Verilog libraries for the Virtex™ device family on the ModelSim™ SE simulator. The compiled results are saved in the default location: `$XILINX/verilog/mti_se.`

```
compxlib –s mti_se –f virtex –l verilog
```

### COMPXLIB Options

This section describes COMPXLIB command line options.

- **–s** (Target Simulator)

  Specify the simulator for which the libraries will be compiled.

  If –s option is not specified, COMPXLIB will exit without compiling the libraries.

  Valid values for –s option are:

  ```
  -s mti_se
  -s mti_pe
  -s ncsim
  -s vcs
  -s vcsi
  -s scirocco
  ```

- **–l** (Language)

  Specify the language from which the libraries will be compiled.

  By default, COMPXLIB detects the language type from the –s (Target Simulator) option. If the simulator supports both Verilog and VHDL, COPMXLIB sets the –l option to *all* and compiles both Verilog and VHDL libraries, otherwise COMXPLIB detects the language type supported by the simulator and sets the –l option value accordingly.

  If the –l option is specified, COMXPLIB compiles the libraries for the language specified with –l option.

  Valid values for –l option are:

  ```
  -l verilog
  -l vhdl
  -l all
  ```

- **-f** (Device Family)

  Specify the device family.

  If –f option is not specified, COMPXLIB will exit with an error message without compiling the libraries.

  Valid values for –f option are,

  ```
  -f all (all device families)
  -f virtex
  -f virtexe
  -f virtex2
  -f virtex2p
  -f spartan2
  -f spartan3
  -f spartan2e
  -f cpld
  -f xpla3
  -f xbr
  -f xc9500
  -f xc9500xl
  -f xc9500xv
  ```

  You can compile selected libraries by using the following –f syntax.

  `–f` *`device_family`*`:`*`descriptor`*`,`*`descriptor`*`,…`

  The *descriptor* identifies the Library Name.

  You can specify the following descriptors:

  | Descriptor | Library |
  |---|---|
  | u | UNISIM |
  | s | SIMPRIM |
  | n | Uni9000 |
  | c | XilinxCoreLib |
  | m | SmartModel |
  | r | CoolRunner™ |
  | a | Abel |

  For example, the following will compile only the Verilog UNISIM library for the Virtex-II Pro™ family on the NC-Sim™ simulator:

  ```
  compxlib -s ncsim -f virtex2p:u –l verilog
  ```

- –**o** (Output Directory)

  Specify the directory path where you want to compile the libraries. By default, COMXPLIB will compile the libraries in `$XILINX/`*`language/target_simulator`* directory.

- –**p** (Simulator Path)

  Specify the directory path where the simulator executables reside. By default, COMPXLIB will automatically search for the path from the `$PATH` environment variable. This option is required if the target simulator is not specified in the `$PATH` environment variable.

- –**w** (Overwrite Compiled Library)

  Specify this option if you want to overwrite the precompiled libraries. By default, COMXPLIB will not overwrite the precompiled libraries.

- –**cfg** (Create Configuration File)

  Specify this option to create a configuration file with default settings. By default, COMPXLIB will create the `compxlib.cfg` file if it is not present in the current directory.

  Use the configuration file to pass run time options to COMPXLIB while compiling the libraries. For more details on the configuration file see "Specifying Run Time Options".

- –**info** (Print Precompiled Library Info)

  Specify this option to print the precompiled information of the libraries. You can specify a directory path with the –info option to print the information for that directory.

- –**help** (Print COMXPLIB help)

  Specify this option to print the COMPXLIB help to standard output.

## COMPXLIB Command Line Examples

- To print the COMPXLIB online help to your monitor screen, type the following at the command line:

  ```
  compxlib -h
  ```

- To compile all of the Verilog libraries for a Virtex™ device (UNISIM, SIMPRIM and XilinxCoreLib) on the ModelSim™ SE simulator and overwrite the results in `$XILINX/verilog/mti_se,` type the following at the command line:

  ```
  compxlib -s mti_se -f virtex -l verilog -w
  ```

- To compile the Verilog UNISIM, Uni9000 and SIMPRIM libraries for the ModelSim™ PE simulator and save the results in the `$MYAREA` directory, type the following at the command line:

  ```
  compxlib -s mti_pe -f all:u:s -l verilog -o $MYAREA
  ```

- To compile the VHDL and Verilog SmartModels for the Cadence™ NC-Sim™ simulator and save the results in `/tmp` directory, type the following at the command line:

  ```
  compxlib -s ncsim -f virtex2p:m -l all -o /tmp
  ```

- To compile the Verilog Virtex-II™ XilinxCoreLib library for the Synopsys® VCS™ simulator and save the results in the default directory, `$XILINX/verilog/vcs,` type the following at the command line:

  ```
  compxlib -s vcs -f virtex2:c
  ```

- To compile the Verilog CoolRunner™ library for the Synopsy®s VCSi™ simulator and save the results in the `current` directory, type the following at the command line:

  ```
  compxlib -s vcsi -f cpld:r -o
  ```

- To compile the Spartan-IIE™ and Spartan-3™ libraries (VHDL UNISIMs, SIMPRIMs and XilinxCoreLib) for the Synopsys® Scirocco™ simulator and save the results in the default directory (`$XILINX/vhdl/scirocco`), and use the simulator executables from the path specified with the –p option, type the following at the command line:

  ```
  compxlib -s scirocco -f spartan2e -p /products/eproduct.ver2_0/2.0/
  comnon/sunos5/bin
  ```

- To print the precompiled library information for the libraries compiled in `$XILINX`, type the following at the command line:

  ```
  compxlib –info
  ```

- To create `compxlib.cfg` with default options, type the following at the command line:

  ```
  compxlib -cfg
  ```

## Specifying Run Time Options

You can specify run time options for COMPXLIB through the `compxlib.cfg` file. By default, COMPXLIB creates this file in the current directory. You can also automatically create this file with its default settings by using the –cfg option.

The following are run time options that can be specified in the configuration file.

- **`EXECUTE ON`** or **`OFF`**

  By default, the value is ON which means that the COMPXLIB compiles the libraries.

  If the value is OFF, COMPXLIB generates only the list of compilation commands in the `compxlib.log` file, without executing them.

- **`LOCK_PRECOMPILED ON`** or **`OFF`**

  By default, the value is OFF which means that the COMPXLIB will compile the dependent libraries automatically if not precompiled.

  If the value is ON, COMPXLIB will not compile the precompiled libraries.

  For example, if you want to compile the SmartModel Library™, COMPXLIB will look for this variable value to see if the dependent libraries, UNISIM and SIMPRIM, are to be compiled or not.

- **`LOG_CMD_TEMPLATE ON`** or **`OFF`**

  By default, the value is OFF which means that the COMPXLIB will not emit the compilation command line in `compxlib.log` file.

  If the value is ON, COMXPLIB will print the compilation commands in the `compxlib.log` file.

- **`PRECOMPILED_INFO ON`** or **`OFF`**

  By default, the value is ON which means that the COMPXLIB will print the precompiled library information including the date the library was compiled.

  If the value is OFF, COMXPLIB will not print the precompiled library information.

- **`OPTION`**

  Simulator Language Command Line Options.

  Syntax:

  ```
  OPTION:Target_Simulator:Language:Command_Line_Options
  ```

By default, COMXPLIB will pick the simulator compilation commands specified in the *Command_Line_Options*.

You can add or remove the options from *Command_Line_Options* depending on the compilation requirements.

### Sample Configuration File:

```
EXECUTE:on
LOCK_PRECOMPILED:off
LOG_CMD_TEMPLATE:on
PRECOMPILED_INFO:on
OPTION:mti_se:vhdl: -source -93
OPTION:mti_se:verilog: -source -93
OPTION:ncsim:vhdl: -MESSAGES -v93 -RELAX -NOLOG
OPTION:ncsim:verilog: -MESSAGES -NOLOG
OPTION:vcsi:verilog: -Mupdate
OPTION:vcs:verilog: -Mupdate
OPTION:scirocco:vhdl: -nc
```

# Running NetGen

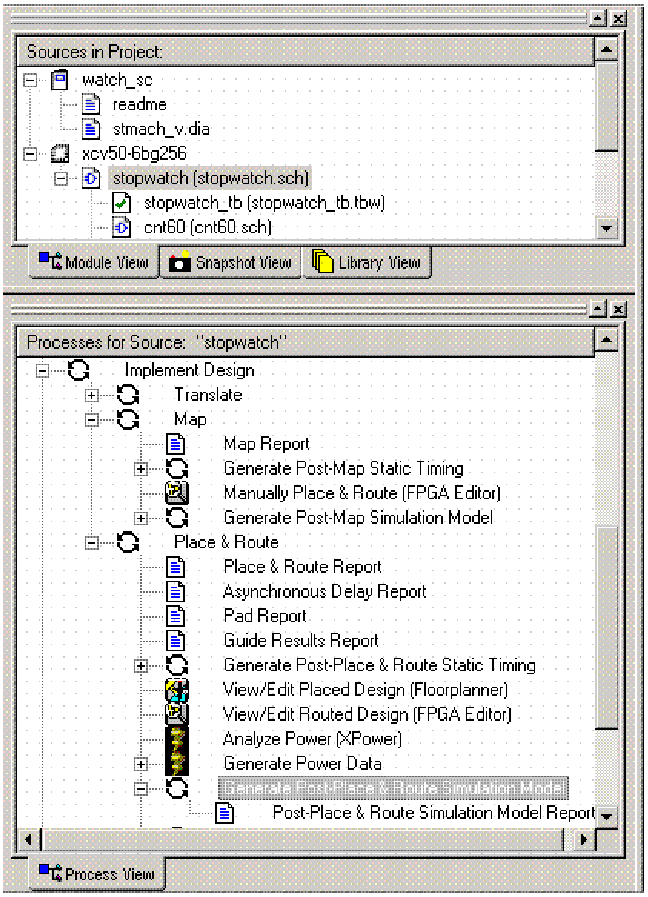
Xilinx® provides a program that can create a verification netlist file from your design files. You can run the netlist writer from Project Navigator, XFLOW, or the command line. Each method is described in the following sections.

## Creating a Simulation Netlist

You can create a timing simulation netlist from Project Navigator, XFLOW, or from the command line, as described in this section.

### From Project Navigator

1. Highlight the top level design in the **Sources in Project** window.
2. In the **Processes for Source** window, click on the “+” sign next to the Implement Design process, and then click on the “+” sign next to the **Place & Route** process.



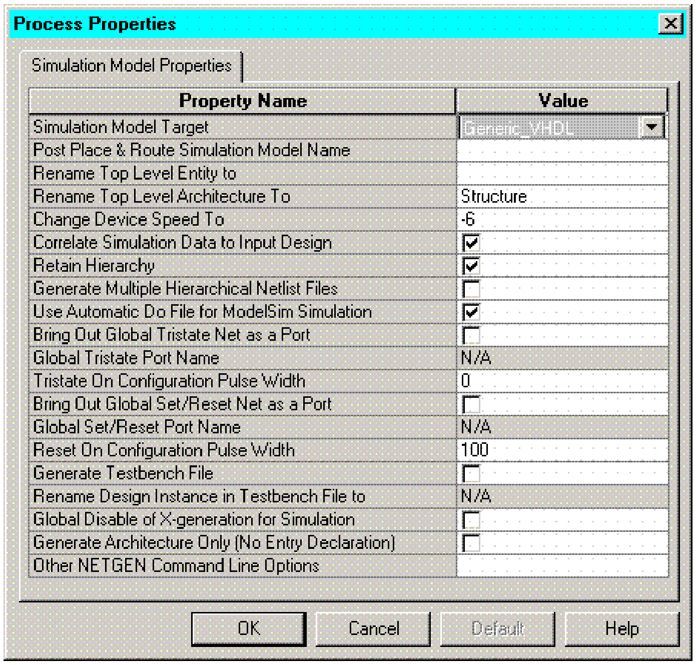
Sources in Project:

- watch_sc
  - readme
  - stmach_v.dia
- xcv50-6bg256
  - stopwatch (stopwatch.sch)
    - stopwatch_tb (stopwatch_tb.tbw)
    - cnt60 (cnt60.sch)

Processes for Source: "stopwatch"

- Implement Design
  - Translate
  - Map
    - Map Report
    - Generate Post-Map Static Timing
    - Manually Place & Route (FPGA Editor)
    - Generate Post-Map Simulation Model
  - Place & Route
    - Place & Route Report
    - Asynchronous Delay Report
    - Pad Report
    - Guide Results Report
    - Generate Post-Place & Route Static Timing
    - View/Edit Placed Design (Floorplanner)
    - View/Edit Routed Design (FPGA Editor)
    - Analyze Power (XPower)
    - Generate Power Data
    - Generate Post-Place & Route Simulation Mode
      - Post-Place & Route Simulation Model Report

3. If any default options need to be changed, right-click on the **Generate Post Place & Route Simulation Model** process and select **Properties**. You can choose the following options from this window:



Process Properties

Simulation Model Properties

| Property Name | Value |
|---|---|
| Simulation Model Target | Generic_VHDL |
| Post Place & Route Simulation Model Name | |
| Rename Top Level Entity to | |
| Rename Top Level Architecture To | Structure |
| Change Device Speed To | -6 |
| Correlate Simulation Data to Input Design | ☑ |
| Retain Hierarchy | ☑ |
| Generate Multiple Hierarchical Netlist Files | □ |
| Use Automatic Do File for ModelSim Simulation | ☑ |
| Bring Out Global Tristate Net as a Port | □ |
| Global Tristate Port Name | N/A |
| Tristate On Configuration Pulse Width | 0 |
| Bring Out Global Set/Reset Net as a Port | □ |
| Global Set/Reset Port Name | N/A |
| Reset On Configuration Pulse Width | 100 |
| Generate Testbench File | □ |
| Rename Design Instance in Testbench File to | N/A |
| Global Disable of X-generation for Simulation | □ |
| Generate Architecture Only (No Entry Declaration) | □ |
| Other NETGEN Command Line Options | |

***Note:*** Project Navigator only shows the options that apply to your specific design flow (i.e if you have created a Verilog project, it only shows you options for creating a Verilog netlist).

- Simulation Model Target

  The Simulation Model Target property allows you to select the target simulator for the simulation netlist. All supported simulators are listed as well as a "generic" VHDL or Verilog netlist for other simulators.

- Post Place & Route Simulation Model Name

  The Post Translate Simulation Model Name property allows you to designate a name for the generated simulation netlist. This only affects the file name for the written netlist and does not affect the entity or module name.

  By default, this field is left blank, and the simulation netlist name is *top_level_name*_timesim.

- Correlate Simulation Data to Input Design

  The Correlate Simulation Data to Input Design property uses an optional NGM file during the back-annotation process. This is a design file produced by Map, that contains information about the original design hierarchy specified by the KEEP_HIERARCHY constraint.

  The default for this property is On (checkbox is checked).

- Generate Multiple Hierarchical Netlist Files

  The Generate Multiple Hierarchical Netlist Files property allows the netlister to generate separate gate-level netlists and SDF files (if applicable) for each hierarchy level in the design that is designated with a KEEP_HIERARCHY attribute. This allows for piece-wise verification of a design. Refer to the "Design Hierarchy and Simulation" section for more details on this option.

The default for this option is Off (checkbox is unchecked). This option must be used with Correlate Simulation Data to Input Design enabled.

- Use Automatic DO File for ModelSim™ Simulation

  This option enables Project Navigator to create and use a batch script file for compiling and simulating your test bench and design files when the ModelSim™ simulator is invoked.

  The default for this option is On (checkbox is checked). This option is only relevant if you invoke the ModelSim™ simulator from Project Navigator.

- Bring Out Global Tristate Net as a Port

  The Bring Out Global Tristate Net as a Port option causes NetGen to bring out the global tristate signal (which forces all FPGA outputs to the high-impedance state) as a port on the top-level design module in the output netlist. Specifying the port name allows you to match the port name you used in the front end if being driven by a TOCBUF. This option should only be used if the global tristate net is not driven by a STARTUP/STARTBUF block. For more information on this option, refer to the "Understanding the Global Reset and Tristate for Simulation" section.

- Global Tristate Port Name

  The Global Tristate Port Name property allows you to specify a port name to match the port name you used in the front end if a TOCBUF component was used.

- Bring Out Global Set/Reset Net as a Port

  The Bring Out Global Set/Reset Net as a Port property causes NetGen to bring out the Global Set/Reset signal (which is connected to all flip-flops and latches in the physical design) as a port on the top-level design module in the output netlist. Specifying the port name allows you to match the port name you used in the front end if a ROCBUF component was used. This option should only be used if the global set/reset net is not driven by a STARTUP/STARTBUF block. For more information on this option, refer to the "Understanding the Global Reset and Tristate for Simulation" section.

- Global Set/Reset Port Name

  The Global Set/Reset Port Name property allows you to specify a port name to match the port name you used in the front end if a ROCBUF component was used.

- Generate Testbench File (VHDL Only)

  The Generate Testbench Template File property creates a test bench file. The file has a .tvhd extension and displays in the "Sources in Project" window.

- Generate Testfixture File (Verilog Only)

  The Generate Testfixture Template File property generates a test fixture file. The file has a .tv extension, and it is a ready-to-use template test fixture that the Verilog file bases on the input design file.

The following options appear if the Advanced Process Settings are enabled in Project Navigator.

- Rename Top Level Entity To (VHDL Only)

  This option allows you to change the name of the top-level entity in the structural VHDL file. By default, the output files inherit the top entity name from the input design file.

- Rename Top Level Module to (Verilog only)

This option allows you to change the name of the top-level module in the structural Verilog file. By default, the output files inherit the top module name from the input design file.

- Rename Top Level Architecture To (VHDL Only)

  This option allows you to rename the architecture name generated by NetGen VHDL. The default architecture name for each entity in the netlist is STRUCTURE.

- Change Device Speed To

  This option allows you to change the targeted speed grade for the output simulation netlist without re-running place and route. This option also allows you to create a simulation netlist with absolute minimum timing numbers if they are available for the target device.

- Retain Hierarchy

  This option, when enabled, allows the netlister to write a verification netlist in which the netlist will retain each level of design hierarchy that was specified on the KEEP_HIERARCHY attribute. When disabled, it removes all hierarchy from the output simulation netlist, and writes out a flat design.

  The default for this option is ON.

- Tristate on Configuration Pulse Width (VHDL Only)

  This option specifies the pulse width, in nanoseconds, for the TOC component. You must specify a positive integer to stimulate the component properly. This option is disabled if you are controlling the global tristate via a port (using the "Bring Out Global Tristate Net as a Port" option). For more information on this option, refer to the “Understanding the Global Reset and Tristate for Simulation” section. By default, the TOC pulse width is set to 0 ns.

- Reset On Configuration Pulse Width (VHDL Only)

  This option specifies the pulse width, in nanoseconds, for the ROC component in the simulation netlist. You must specify a positive integer to stimulate the component properly. This option is disabled if you are controlling the global reset via a port (using the "Bring Out Global Set/Reset Net as a Port" option). For more information on this option, see the “Understanding the Global Reset and Tristate for Simulation” section. By default, the ROC pulse width is set to 100 ns.

- Rename Design Instance in Testbench File To

  This option specifies the name of the top-level design instance name appearing within the output test bench template file if the "Generate Testbench/Testfixture File" option is selected. The option allows you to match the top-level instance name to the name specified in your RTL test bench file. The default name for the test bench instance is UUT.

- Include 'uselib Directive in Verilog File (Verilog Only)

  The Include 'uselib Directive in Verilog File property causes ISE to write a library path pointing to the SIMPRIM library into the output Verilog (.v) file. In general, Xilinx® only suggests that you use this option with the Verilog-XL simulator when simulations are performed on the same network as where the ISE software exists. By default, this field is set to Off (checkbox is blank).

- Path Used in $SDF_annotate (Verilog Only)

  This option allows you to specify a path to the SDF file that you want written to the `$sdf_annotate` function in the Verilog netlist file. If a full path is not specified, it writes the full path of the current work directory and the SDF file name to the `$sdf_annotate` file.

The default path for the SDF file is in the same directory in which the Verilog simulation netlist resides.

- Global Disable of X-generation for Simulation (VHDL Only)

  This option is used to disable X-generation by all registers in the design when a timing violation occurs. If this option is set, all registers in the design retain their last value when a timing violation occurs. For more information on this option, refer to "Disabling 'X' Propagation" in this manual. The default value for this option is OFF.

- Generate Architecture Only (No Entry Declaration)

  Specifies whether to create an entity for each level of hierarchy in the design, or whether to generate the architecture portion only. This option is useful when generics are declared in the top-level entity declaration in the original RTL design as it allows the re-use of the original entity declaration for proper linking of the structural design to the test bench file. By default, this property is set to False (checkbox is unchecked), and both entity and architectures are created in the resulting netlist.

- Other NetGen Command Line Options

  This allows the user to specify options for NetGen that are not available from the above options.

4. Double-click **Generate Post Place & Route Simulation Model**. Project Navigator now runs through the steps required to produce the back-annotated simulation netlist.

## From XFLOW

To display the available options for XFLOW, and for a complete list of the XFLOW option files, type "flow" at the prompt without any arguments. For complete descriptions of the options and the option files, see the *Development System Reference Guide*.

1. Open a command terminal and change directory to the project directory.
2. Type the following at the command prompt:
   - To create a functional simulation (Post NGD) netlist from an input design EDIF file:

     ```
     > xflow -fsim option_file.opt design_name.edif
     ```

   - To create a timing simulation (post PAR) netlist from an input EDIF design file:

     ```
     > xflow -implement option_file -tsim option_file design_name.edf
     ```

   - To create a timing simulation (Post PAR) netlist from an NCD file:

     ```
     > xflow -tsim option_file.opt design_name.ncd
     ```

XFLOW runs the appropriate programs with the options specified in the option file. To change the options, run xflow first with the –norun switch to have XFLOW copy the option file(s) to the project directory. Then edit the appropriate option file to modify the run parameters for the flow. For more information on running XFLOW, see the *Development System Reference Guide*.

### From Command Line or Script File

- Post-NGD simulation

  To run a post-NGD simulation, type the following at the command line:

  For Verilog:

  ```
  netgen -sim -ofmt verilog [options] design.ngd
  ```

  For VHDL:

  ```
  netgen -sim -ofmt vhdl [options] design.ngd
  ```

- Post-Map simulation

  To run a post-Map simulation, perform the following command line operations:

  ```
  ngdbuild options design
  map options design.ngd
  ```

  For Verilog:

  ```
  netgen -sim -ofmt verilog [options] design.ncd -ngm design.ngm
  ```

  For VHDL:

  ```
  netgen -sim -ofmt vhdl [options] design.ncd -ngm design.ngm
  ```

- Post-PAR simulation

  To run a post-PAR simulation, perform the following command line operations:

  ```
  ngdbuild options design
  map options design.ngd
  par options design.ncd -w design_par.ncd
  ```

  For Verilog:

  ```
  netgen -sim -ofmt verilog [options] design_par.ncd -ngm
  design.ngm
  ```

  For VHDL:

  ```
  netgen -sim -ofmt vhdl [options] design_par.ncd -ngm design.ngm
  ```

## Disabling ‘X’ Propagation

During a timing simulation, when a timing violation occurs, the default behavior of a latch, register, RAM or other synchronous element is to output an ’X’ to the simulator. The reason for this is that when a timing violation occurs, it is not known what the actual output value should be. The output of the register could retain its previous value, update to the new value, or perhaps go metastable in which a definite value is not settled upon until sometime after the clocking of the synchronous element. Since this value cannot be determined, accurate simulation results cannot be guaranteed, and so the element outputs an ’X’ to represent an unknown value. The ’X’ output remains until the next clock cycle in which the next clocked value updates the output if another violation does not occur.

Sometimes this situation can have a drastic affect on simulation. For example, an ’X’ generated by one register can be propagated to others on subsequent clock cycles, causing large portions of the design under test to become ’unknown’. If this happens on a synchronous path in the design, you can ensure a properly operating circuit by analyzing the path and fixing any timing problems associated with this or other paths in the design. If however, this path is an asynchronous path in the design, and you cannot avoid timing violations, you can disable the ’X’ propagation on synchronous elements during timing violations, so that these elements do not output an ’X’. When ’X’ propagation is disabled, the previous value is retained at the output of the register. Please understand that in the actual silicon, the register may have very well changed to the ’new’ value, and that disabling ’X’ propagation may yield simulation results that do not match the silicon behavior. Exercise caution when using this option; you should only use it when you cannot otherwise avoid timing violations.

## Using the ASYNC_REG Attribute

ASYNC_REG is a constraint in the Xilinx® software that helps identify asynchronous registers in the design and disable 'X' propagation for those particular registers. If the attribute ASYNC_REG is attached to a register in the front end design by either an attribute in HDL code or by a constraint in the UCF, during timing simulation, those registers retain the previous value, and do not output an 'X' to simulation. A timing violation error should still occur, so use caution as the new value may have been clocked in very well. The ASYNC_REG attribute is only applicable to CLB and IOB registers and latches.

If clocking in asynchronous data cannot be avoided, Xilinx® suggests that you only do so on IOB or CLB registers. Clocking in asynchronous signals to RAM, SRL or other synchronous elements has less deterministic results, and therefore should be avoided. Refer to the *Constraints Guide* for more information on using the ASYNC_REG constraint. Xilinx® highly suggests to first properly synchronize any asynchronous signal in a register, latch or FIFO before writing to a RAM, SRL or any other synchronous element.

## Using Global Switches

Xilinx® generally suggests that you use the ASYNC_REG to control 'X' propagation. However, there is an option to use use global switches to disable 'X' propagation for all components in the simulation.

### Verilog

For Verilog, use the +no_notifier switch from within your simulator. When a timing violation occurs, the simulator puts out a message, but the synchronous element retains its previous value. Please consult your simulator's documentation for details on applying and using the +no_notifier switch.

### VHDL

For VHDL if the simulator does not have a switch to disable 'X' propagation, NetGen can create a netlist in which this behavior is disabled. By invoking NetGen with the –xon FALSE switch, the previous value should be retained during a timing violation. If the simulation netlist is created within the ISE environment, use the "Global Disable of X-generation for Simulation" option in the advanced process properties options for Generate Post-Map Simulation Model.

## Use With Care

Xilinx® highly recommends that you only disable ‘X’ propagation on paths that are truly asynchronous where it is impossible to meet synchronous timing requirements. This capability is present for simulation in the event that timing violations cannot be avoided, such as when a register must input asynchronous data. Use extreme caution when disabling 'X' propagation as simulation results may no longer properly reflect what is happening in the silicon.

## MIN/TYP/MAX Simulation

The Standard Delay Format (SDF) file allows you to specify three sets of delay values for simulation. These are Minimum, Typical, and Maximum (worst case), typically abbreviated as MIN:TYP:MAX. Xilinx® however, only writes a single value for all three fields in the SDF file for simulation. By default, Xilinx® uses the worst case values for the speed grade of the target architecture at the maximum operating temperature, the minimum voltage, and various process variations to populate the MAX, TYP, and MIN delay sets in the SDF file. This value set is used for most timing simulation runs to test circuit operation and timing.

Optionally, NetGen can produce absolute minimum delay values for simulation by applying the –s min switch. The resulting SDF file produced from NetGen has the absolute process minimums populated in all three SDF fields: MIN, TYP and MAX. Absolute process MIN values are the absolute fastest delays that a path can run in the target architecture given the best operating conditions within the specifications of the architecture: lowest temperature, highest voltage, best possible silicon. Generally, these process minimum delay values are only useful for checking board-level, chip-to-chip timing for high-speed data paths in best / worst case conditions.

By default, the worst case delay values are derived from the worst temperature, voltage, and silicon process for a particular target architecture. If better temperature and voltage characteristics can be ensured during the operation of the circuit, you can use prorated worst case values in the simulation to gain better performance results. The default would apply worst case timing values over the specified TEMPERATURE and VOLTAGE within the operating conditions recommended for the device.

Prorating is a linear scaling operation. It applies to existing speed file delays, and is applied globally to all delays. The prorating constraints, VOLTAGE and TEMPERATURE, provide a method for determining timing delay characteristics based on known environmental parameters.

The VOLTAGE constraint provides a means of prorating delay characteristics based on the specified voltage applied to the device. The UCF syntax is as follows:

```
VOLTAGE=value[V]
```

Where value is an integer or real number specifying the voltage and units is an optional parameter specifying the unit of measure.

The TEMPERATURE constraint provides a means of prorating device delay characteristics based on the specified junction temperature. The UCF syntax is as follows:

```
TEMPERATURE=value[C|F|K]
```

Where value is an integer or a real number specifying the temperature. C, K, and F are the temperature units: F is degrees Fahrenheit, K is degrees Kelvin, and C is degrees Celsius, the default.

The resulting values in the SDF fields when using prorated TEMPERATURE and / or VOLTAGE values are prorated worst case values for the MIN, TYP and MAX fields.

Refer to the specific product Data Sheets to determine the specific range of valid operating temperatures and voltages for the target architecture. If the temperature or voltage specified in the constraint does not fall within the supported range, the constraint is ignored and an architecture specific default value is used instead. Not all architectures support prorated timing values. For simulation, the VOLTAGE and TEMPERATURE constraints are processed from the UCF file into the PCF file. The PCF file must then be referenced when running NetGen in order to pass the operating conditions to the delay annotator.

To generate a simulation netlist using prorating for VHDL, type the following:

```
netgen -sim -ofmt vhdl [options] -pcf design.pcf design.ncd
```

To generate a simulation netlist using prorating for Verilog, type the following

```
netgen -sim -ofmt verilog [options] -pcf design.pcf design.ncd
```

***Note:*** Combining both minimum values would override prorating, and result in issuing only absolute process MIN values for the simulation SDF file. Prorating may only be available for select FPGA families, and it is not intended for military and industrial ranges. It is applicable only within the commercial operating ranges.

| NetGen Option | MIN:TYP:MAX Field in SDF File Produced by NetGen –sim |
|---|---|
| default | MAX:MAX:MAX |
| –s min | Process MIN: Process MIN: Process MIN |
| Prorated voltage / temperature in UCF / PCF | MAX: Prorated: MAX: Prorated MAX |

## Understanding the Global Reset and Tristate for Simulation

Xilinx® FPGAs have dedicated routing and circuitry that connects to every register in the device. The dedicated global GSR (Global Set-Reset) net is asserted and is released during configuration immediately after the device is configured. All the flip-flops and latches receive this reset and are either set or reset, depending on how the registers are defined.

Although designers can access the GSR net after configuration, Xilinx does not recommend using the GSR circuitry in place of a manual reset. This is because the FPGAs offer high-speed backbone routing for high fanout signals like a system reset. This backbone route is faster than the dedicated GSR circuitry and is easier to analyze than the dedicated global routing that transports the GSR signal.

In back-end simulations, a GSR signal is automatically pulsed for the first 100 ns to simulate the reset that occurs after configuration. A GSR pulse can optionally be supplied in front end functional simulations, but is not necessary if the design has a local reset that resets all registers. When creating a test bench, it is important to remember that the GSR pulse occurs automatically in the back-end simulation, as this will hold all registers in reset for the first 100 ns of the simulation. For more information about controlling the GSR pulse or inserting a GSR pulse in the front end simulation, see the “Simulating VHDL” and “Simulating Verilog” sections.

In addition to the dedicated global GSR, all output buffers are set to a high impedance state during configuration mode with the dedicated GTS (global output tristate enable) net. All general-purpose outputs are affected whether they are regular, tristate, or bi-directional outputs during normal operation. This ensures that the outputs do not erroneously drive other devices as the FPGA is being configured.

In simulation, the GTS signal is usually not driven. The circuitry for driving GTS is available in the back-end simulation and can be optionally added for the front end simulation, but the GTS pulse width is set to 0 by default. For more information about controlling the GTS pulse or inserting the circuitry to pulse GTS in the front end simulation, see the “Simulating VHDL” and “Simulating Verilog” sections.

The following figure shows how the global GTS and GSR signals are used in the FPGA.

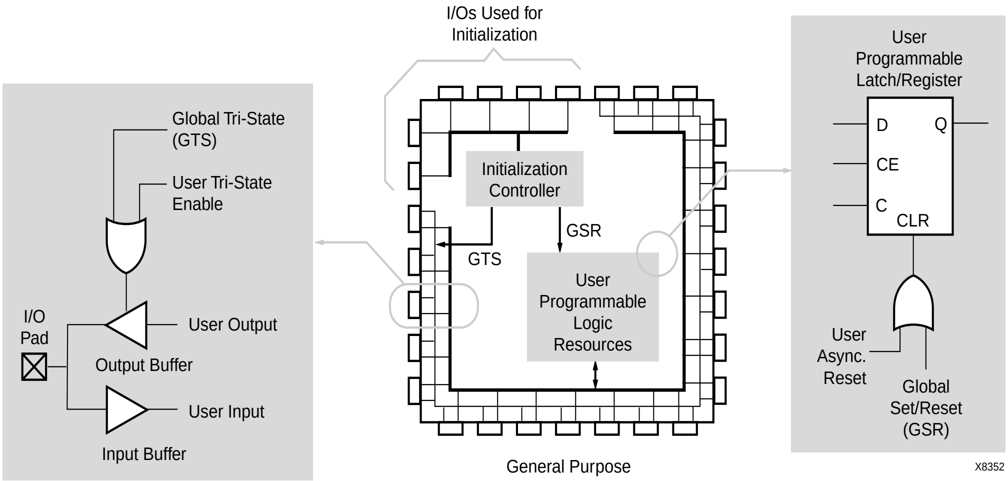


*Figure 6-2:* **Built-in FPGA Initialization Circuitry**

# Simulating VHDL

## Emulating the Global GSR pulse in VHDL in Functional Simulation

Many HDL designs targeted for Xilinx® FPGAs have a user reset that initializes all registers in the design during the functional simulation. For these designs, it is not necessary to emulate the GSR pulse in the functional simulation. If the design contains registers that are not connected to a user reset, the GSR pulse can be emulated to ensure that the functional simulation matches the timing simulation. There are two methods that can be used to emulate the GSR pulse:

- Use the ROC cell to generate a one-time GSR pulse at the beginning of the simulation as described in the "Using VHDL Reset-On-Configuration (ROC) Cell" section.
- Use the ROCBUF cell to control the emulated GSR signal in the test bench as described in the "Using VHDL ROCBUF Cell" section.

### Using VHDL Reset-On-Configuration (ROC) Cell

The ROC cell, which is modeled in the UNISIM library, can be used to emulate the GSR pulse at the beginning of a functional simulation. This is the same component that is automatically inserted into the back-end netlist. It generates a one time pulse at the beginning of the simulation that lasts for a default value of 100ns.

During implementation, the signal connected to the output of the ROC component will automatically be mapped to the Global GSR network and will not be routed on local routing.

Following is an example that shows how to use the ROC cell.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
library UNISIM;
use UNISIM.all;
entity EX_ROC is
  port (
        CLOCK, ENABLE : in std_logic;
        CUP, CDOWN : out std_logic_vector (3 downto 0)
        );
end EX_ROC;
architecture A of EX_ROC is
  signal GSR : std_logic;
  signal COUNT_UP, COUNT_DOWN : std_logic_vector (3 downto 0);
  component ROC
    port (O : out std_logic);
  end component;
begin
  U1 : ROC port map (O => GSR);

  UP_COUNTER : process (CLOCK, ENABLE, GSR)
    begin
      if (GSR = ’1’) then
          COUNT_UP <= "0000";
      elsif (CLOCK’event AND CLOCK = ’1’) then
        if (ENABLE = ’1’) then
            COUNT_UP <= COUNT_UP + "0001";
        end if;
      end if;
  end process UP_COUNTER;
  DOWN_COUNTER : process (CLOCK, ENABLE, GSR, COUNT_DOWN)
  begin
    if (GSR = ’1’ OR COUNT_DOWN = "0101") then
        COUNT_DOWN <= "1111";
    elsif (CLOCK’event AND CLOCK = ’1’) then
        if (ENABLE = ’1’) then
            COUNT_DOWN <= COUNT_DOWN - "0001";
        end if;
    end if;
  end process DOWN_COUNTER;
  CUP <= COUNT_UP;
  CDOWN <= COUNT_DOWN;
end A;
```

The following figure shows the progression of the ROC model and its interpretation in the four main design phases.

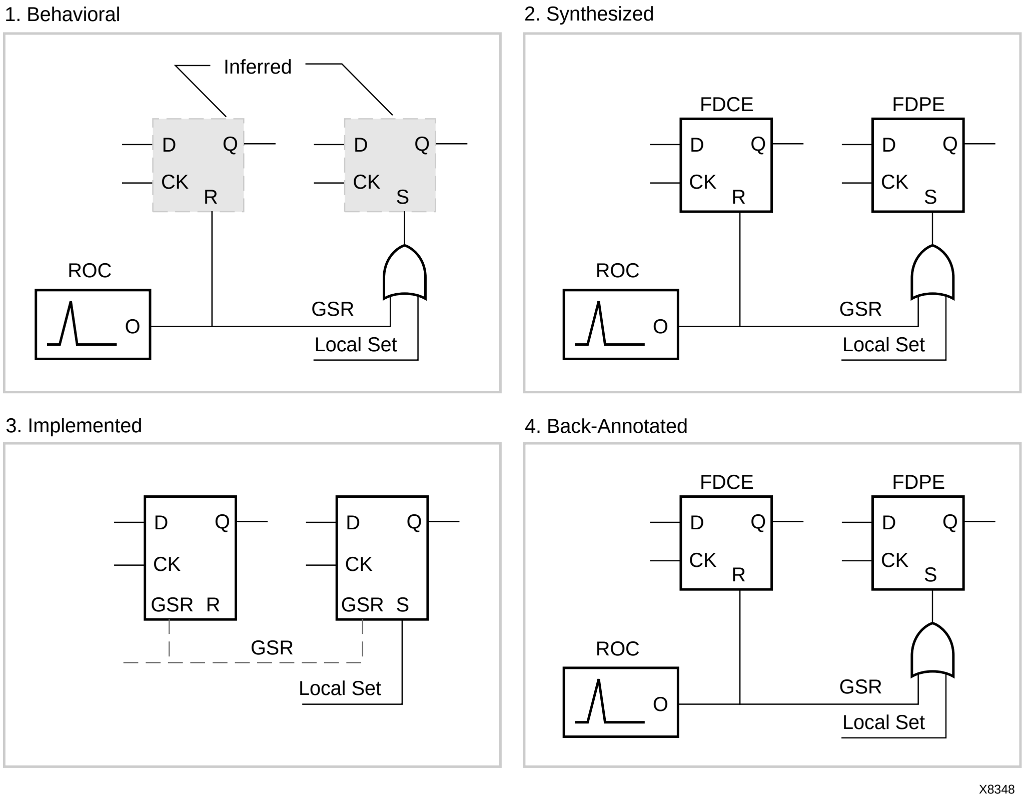


*Figure 6-3:* **ROC Simulation and Implementation**

- *Behavioral Phase*—In this phase, the behavioral or RTL description registers are inferred from the coding style, and the ROC cell is instantiated. This ensures that GSR behavior at the RTL level matches the behavior of the post-synthesis and implementation netlists.
- *Synthesized Phase*—In this phase, inferred registers are mapped to a technology and the ROC instantiation is carried from the RTL to the implementation tools. As a result, consistent global set / reset behavior is maintained between the RTL and synthesized structural descriptions during simulation.
- *Implemented Phase*—During implementation, the ROC is removed from the logical description and is placed and routed as a pre-existing circuit on the chip. All set / resets for the registers are automatically assumed to be driven by the global set / reset net so data is not lost.
- *Back-annotated Phase*—In this phase, the Xilinx® VHDL netlist program assumes all registers are driven by the GSR net; uses the X_ROC simulation model for the ROC; and rewires it to the GSR nets in the back-annotated netlist. For a non-hierarchical netlist, the GSR net is a fully wired net and the X_ROC cell drives it. For a hierarchical netlist, the GSR net is not wired across hierarchical modules. The GSR net in each hierarchical module is driven by an X_ROC cell. The ROC pulse width of the X_ROC component can be controlled using the -rpw switch for NetGen.

## Using VHDL ROCBUF Cell

A second method of emulating GSR in the functional simulation is to use the ROCBUF. This component creates a buffer for the global set/reset signal, and provides an input port on the buffer to drive the global set reset line. This port must be declared in the entity list and driven in RTL simulation.

This method is applicable when system-level issues make your design's initialization synchronous to an off-chip event. In this case, you provide a pulse that initializes your design at the start of simulation time, and you possibly provide further pulses as simulation time progresses (perhaps to simulate cycling power to the device).

During the place and route process, this port is removed; it is not implemented on the chip. ROCBUF does not by default reappear in the post-routed netlist unless the –gp switch is used during NetGen netlisting. The nets driven by a ROCBUF must be an active High set/reset.

The following example illustrates how to use the ROCBUF in your designs.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

library UNISIM;
use UNISIM.all;

entity EX_ROCBUF is
  port (
      CLOCK, ENABLE, SRP : in std_logic;
      CUP, CDOWN : out std_logic_vector (3 downto 0)
      );
end EX_ROCBUF;

architecture A of EX_ROCBUF is
  signal GSR : std_logic;
  signal COUNT_UP, COUNT_DOWN : std_logic_vector (3 downto 0);
  component ROCBUF
    port (
      I : in std_logic;
      O : out std_logic
      );
  end component;
begin
  U1 : ROCBUF port map (I => SRP, O => GSR);
  UP_COUNTER : process (CLOCK, ENABLE, GSR)
  begin
    if (GSR = ’1’) then
      COUNT_UP <= "0000";
    elsif (CLOCK’event AND CLOCK = ’1’) then
      if (ENABLE = ’1’) then
        COUNT_UP <= COUNT_UP + "0001";
      end if;
    end if;
  end process UP_COUNTER;
```

```
    DOWN_COUNTER : process (CLOCK, ENABLE, GSR, COUNT_DOWN)
    begin
      if (GSR = '1' OR COUNT_DOWN = "0101") then
          COUNT_DOWN <= "1111";
      elsif (CLOCK'event AND CLOCK = '1') then
          if (ENABLE = '1') then
              COUNT_DOWN <= COUNT_DOWN - "0001";
          end if;
      end if;
    end process DOWN_COUNTER;
    CUP <= COUNT_UP;
    CDOWN <= COUNT_DOWN;
  end A;
```

The following figure shows the progression of the ROCBUF model and its interpretation in the four main design phases.

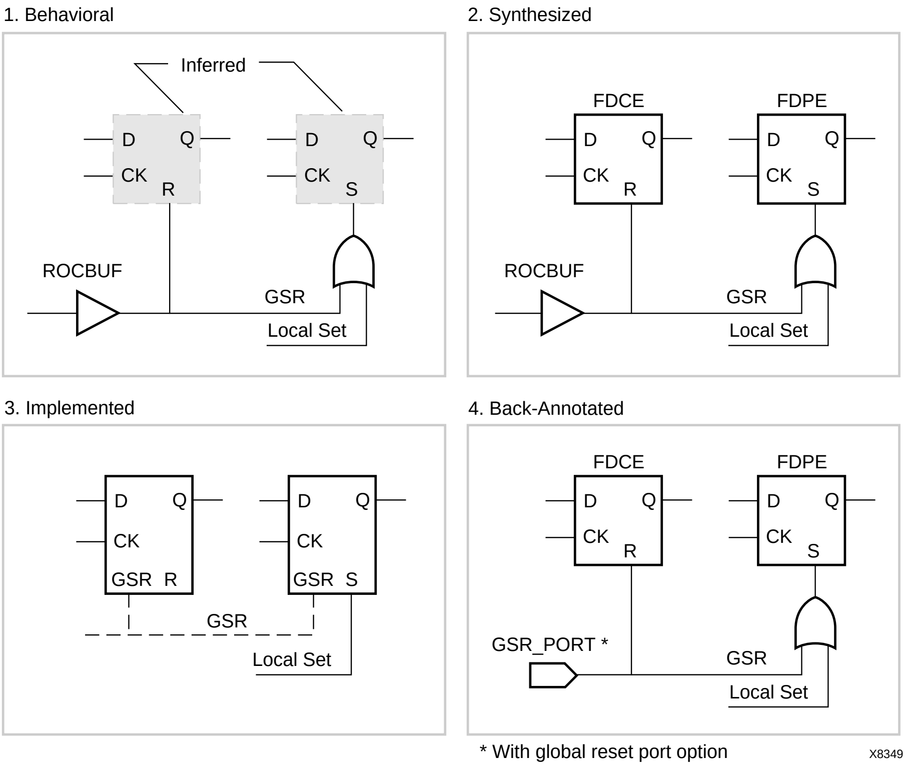


*Figure 6-4:* **ROCBUF Simulation and Implementation**

- *Behavioral Phase*—In this phase, the behavioral or RTL description registers are inferred from the coding style, and the ROCBUF cell is instantiated. Use the ROCBUF cell instead of the ROC cell when you want test bench control of GSR simulation.
- *Synthesized Phase*—In this phase, inferred registers are mapped to a technology and the ROCBUF instantiation is carried from the RTL to the implementation tools. As a result, consistent global set / reset behavior is maintained between the RTL and synthesized structural descriptions during simulation.
- *Implemented Phase*—During implementation, the ROCBUF is removed from the logical description of the design and the global resources are used for the set / reset function.

- *Back-annotated Phase*—In this phase, use the NetGen option –gp to replace the port that was previously occupied by the ROCBUF in the RTL description of the design. A non-hierarchical netlist will have this port drive the fully wired GSR signal. For hierarchical designs, the GSR net is not wired across hierarchical modules. The VHDL global signal, X_GSR_GLOBAL_SIGNAL, is used to connect the top level port created by the –gp switch to the GSR signals of the hierarchical modules. Each hierarchical module contains an X_ROCBUF cell, which is driven by X_GSR_GLOBAL_SIGNAL. The output of the X_ROCBUF is connected to the GSR signal of the hierarchical module. Toggling the input to the top level port affects the entire design. The X_GSR_GLOBAL_SIGNAL global signal is defined in a package within the VHDL netlist. The package name is *design_name*_ROCTOC.

  If desired, each of the hierarchical modules can be written out as a separate netlist using the NetGen –mhf switch to create multiple hierarchical files. This helps in analyzing lower level modules individually. Every netlist contains the *design_name*_ROCTOC package definition with X_GSR_GLOBAL_SIGNAL signal. When lower modules are simulated, though X_ROCBUF is connected to X_GSR_GLOBAL_SIGNAL, there will be no event on it. Hence the X_ROCBUF creates a pulse similar to X_ROC and the GSR of the module is toggled. The ROC pulse width of the X_ROCBUF component can be controlled using the –rpw switch for NetGen.

  If all the modules, including the top level design module, are compiled and analyzed together, the top level port created via the -gp switch would drive the X_GSR_GLOBAL_SIGNAL and all X_ROCBUF cells. In this situation, the X_ROCBUF component does not create a pulse, but rather behaves like a buffer passing on the value of the X_GSR_GLOBAL_SIGNAL. The ROC pulse width of the X_ROCBUF component can be controlled using the -rpw switch for NetGen.

## Using VHDL STARTBUF_VIRTEX, STARTBUF_VIRTEX2 Block or the STARTBUF_SPARTAN2 Block

The STARTUP_VIRTEX, STARTUP_VIRTEX2 and STARTUP_SPARTAN2 blocks can be instantiated to identify the GSR signals for implementation if the global reset or tristate is connected to a chip pin. However, these cells cannot be simulated as there is no simulation model for them.

*Table 6-5:* **Virtex™/E and Spartan-II™ STARTBUF/STARTUP Pins**

| STARTBUF Pin Names | Connection Points | Virtex™/E STARTUP Pin Names | Spartan-II™ STARTUP Pin Names |
|---|---|---|---|
| GSRIN | Global Set/Reset Port of Design | GSR | GSR |
| GTSIN | Global Tristate Port of Design | GTS | GTS |
| CLKIN | Port or Internal Logic | CLK | CLK |

*Table 6-5:* **Virtex™/E and Spartan-II™ STARTBUF/STARTUP Pins**

| STARTBUF Pin Names | Connection Points | Virtex™/E STARTUP Pin Names | Spartan-II™ STARTUP Pin Names |
|---|---|---|---|
| GTSOUT | All Output Buffers Tristate Control | N/A | N/A |
| GSROUT | All Registers Asynchronous set/reset | N/A | N/A |

Xilinx® recommends that you use the local routing for Virtex™ devices as opposed to using the dedicated GSR. If the design resources are available, this method gives better performance and more predictable design behavior.

If you do not plan to bring the GSR pin out to a device pin, but want to have access to it for simulation, Xilinx® suggests that you use the ROC or ROCBUF.

## Emulating the Global GTS pulse in a VHDL Functional Simulation

In most cases it is not necessary to emulate the global GTS pulse, but if pre-configuration behavior of the device I/Os needs to be incorporated into the functional simulation, you can use the following:

- Use the TOC cell to generate a one-time GTS pulse at the beginning of the simulation as described in the "Using VHDL Tristate-On-Configuration (TOC)" section.
- Use the TOCBUF cell and control the emulated GTS signal in the test bench as described in the "Using VHDL TOCBUF" section.

### Using VHDL Tristate-On-Configuration (TOC)

The TOC cell, which is modeled in the UNISIM library, can be used to emulate the GTS pulse at the beginning of a functional simulation. This is the same component that is automatically inserted into the back-end netlist. It generates a one-time pulse at the beginning of the simulation that lasts for a default value of 100ns. The pulse width is a generic that can be passed to the TOC model to change the pulse width.

***Note:*** The default pulse width in the UNISIM TOC model is 100 ns, but the default pulse width in the back-end netlist is 0. When using the TOC cell in a functional simulation, the –tpw switch can be used in NetGen to change the pulse width to match the functional simulation.

During implementation, the signal connected to the output of the TOC component will automatically be mapped to the Global GTS network and will not be routed on local routing.

The following is an example of how to use the TOC cell.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
library UNISIM;
use UNISIM.all;
entity EX_TOC is
 port (
    CLOCK, ENABLE : in std_logic;
    CUP, CDOWN : out std_logic_vector (3 downto 0)
    );
end EX_TOC;
architecture A of EX_TOC is
 signal GSR, GTS : std_logic;
 signal COUNT_UP, COUNT_DOWN : std_logic_vector (3 downto 0);
 component ROC
  port (O : out std_logic);
 end component;
 component TOC
  port (O : out std_logic);
 end component;
begin
 U1 : ROC port map (O => GSR);
 U2 : TOC port map (O => GTS);
 UP_COUNTER : process (CLOCK, ENABLE, GSR)
 begin
  if (GSR = ’1’) then
    COUNT_UP <= "0000";
  elsif (CLOCK’event AND CLOCK = ’1’) then
    if (ENABLE = ’1’) then
      COUNT_UP <= COUNT_UP + "0001";
    end if;
  end if;
 end process UP_COUNTER;
 DOWN_COUNTER : process (CLOCK, ENABLE, GSR, COUNT_DOWN)
 begin
  if (GSR = ’1’ OR COUNT_DOWN = "0101") then
    COUNT_DOWN <= "1111";
  elsif (CLOCK’event AND CLOCK = ’1’) then
    if (ENABLE = ’1’) then
      COUNT_DOWN <= COUNT_DOWN - "0001";
    end if;
  end if;
 end process DOWN_COUNTER;
 CUP <= COUNT_UP when (GTS = ’0’ AND COUNT_UP /= "0000") else "ZZZZ";
 CDOWN <= COUNT_DOWN when (GTS = ’0’) else "ZZZZ";
end A;
```

The following figure shows the progression of the TOC model and its interpretation in the four main design phases.

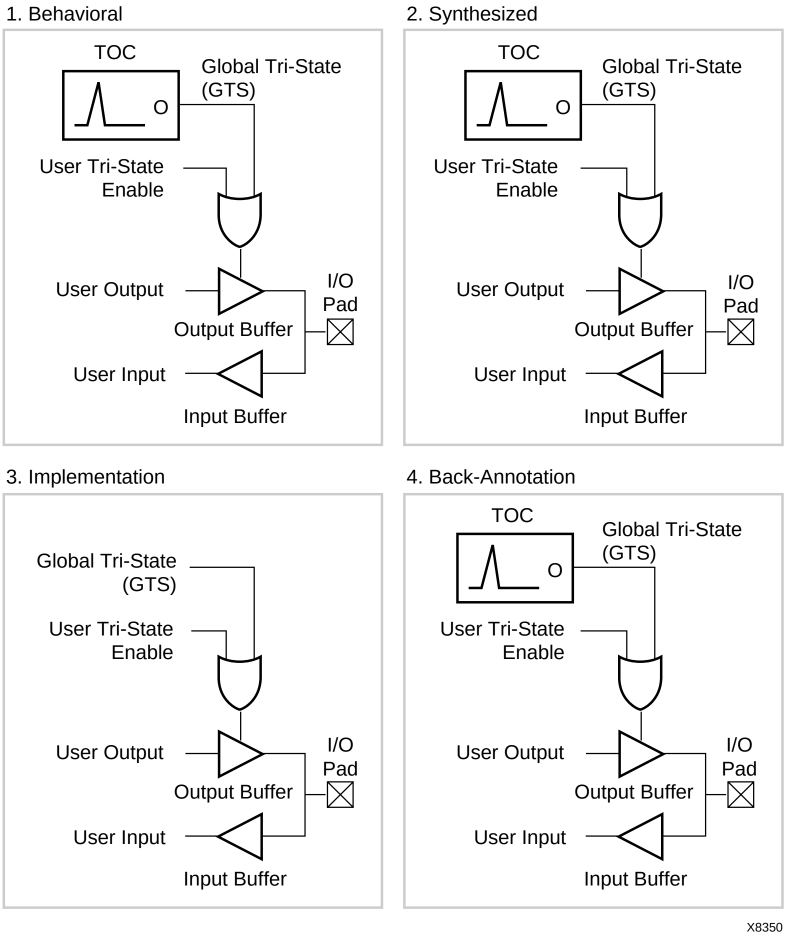


*Figure 6-5:* **TOC Simulation and Implementation**

- *Behavioral Phase*—In this phase, the behavioral or RTL description of the output buffers is inferred from the coding style. The TOC cell can be instantiated and connected to all tristate outputs in the design.
- *Synthesized Phase*—In this phase, the inferred I/Os are mapped to a device, and the TOC instantiation is carried from the RTL to the implementation tools. This results in maintaining consistent global output tristate enable behavior between the RTL and the synthesized structural descriptions during simulation.
- *Implemented Phase*—During implementation, the TOC is removed from the logical description and the global tristate net resource is used.
- *Back-annotation Phase*—In this phase, the Xilinx® VHDL netlist program assumes all registers are driven by the GTS net; uses the X_TOC simulation model for the TOC; and rewires it to the GTS nets in the back-annotated netlist. For a non-hierarchical netlist, the GTS net is a fully wired net and the X_TOC cell drives it. For a hierarchical netlist, the GTS net is not wired across hierarchical modules. The GTS net in each hierarchical module is driven by an X_TOC cell. The TOC pulse width of the X_TOC component can be controlled using the -tpw switch for NetGen.

## Using VHDL TOCBUF

A second method of emulating GTS in the functional simulation is to use the TOCBUF. This component creates a buffer for the global GTS net, and provides an input port on the buffer to drive GTS. This port must be declared in the entity list and driven in RTL simulation.

This method is applicable when system-level issues make your design's initialization synchronous to an off-chip event. In this case, you provide a pulse that tristates the outputs at the start of simulation time, and you possibly provide further pulses as simulation time progresses (perhaps to simulate cycling power to the device).

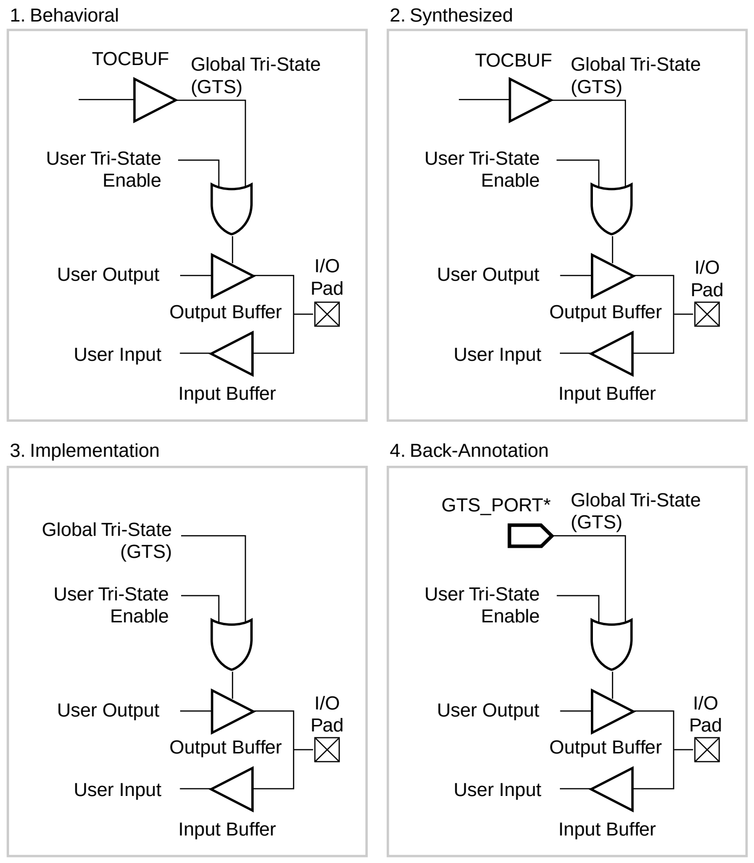
During the place and route process, this port is removed; it is not implemented on the chip. TOCBUF does not by default reappear in the post- routed netlist unless the –tp switch is used in NetGen.

The following example illustrates how to use the TOCBUF in your design.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
library UNISIM;
use UNISIM.all;
entity EX_TOCBUF is
  port (
      CLOCK, ENABLE, SRP, STP : in std_logic;
      CUP, CDOWN : out std_logic_vector (3 downto 0)
      );
end EX_TOCBUF;
architecture A of EX_TOCBUF is
  signal GSR, GTS : std_logic;
  signal COUNT_UP, COUNT_DOWN : std_logic_vector (3 downto 0);
  component ROCBUF
    port (
      I : in std_logic;
      O : out std_logic
      );
  end component;
  component TOCBUF
    port (
      I : in std_logic;
      O : out std_logic
      );
  end component;
begin
  U1 : ROCBUF port map (I => SRP, O => GSR);
  U2 : TOCBUF port map (I => STP, O => GTS);
  UP_COUNTER : process (CLOCK, ENABLE, GSR)
  begin
    if (GSR = ’1’) then
      COUNT_UP <= "0000";
    elsif (CLOCK’event AND CLOCK = ’1’) then
      if (ENABLE = ’1’) then
        COUNT_UP <= COUNT_UP + "0001";
      end if;
    end if;
  end process UP_COUNTER;
```

```
 DOWN_COUNTER : process (CLOCK, ENABLE, GSR, COUNT_DOWN)
 begin
  if (GSR = ’1’ OR COUNT_DOWN = "0101") then
    COUNT_DOWN <= "1111";
  elsif (CLOCK’event AND CLOCK = ’1’) then
    if (ENABLE = ’1’) then
      COUNT_DOWN <= COUNT_DOWN - "0001";
    end if;
  end if;
 end process DOWN_COUNTER;
 UP <= COUNT_UP when (GTS = ’0’ AND COUNT_UP /= "0000") else "ZZZZ";
 CDOWN <= COUNT_DOWN when (GTS = ’0’) else "ZZZZ";
end A;
```

The following figure shows the progression of the TOCBUF model and its interpretation in the four main design phases.



* with global tri-state port option

*Figure 6-6:* **TOCBUF Simulation and Implementation**

- *Behavioral Phase*—In this phase, the behavioral or RTL description of the output buffers is inferred from the coding style and may be inserted. You can instantiate the TOCBUF cell.
- *Synthesized Phase*—In this phase, the inferred output buffers are mapped to a device and the TOCBUF instantiation is carried from the RTL to the implementation tools. This maintains consistent global output tristate enable behavior between the RTL and the synthesized structural descriptions during simulation.

- *Implemented Phase*—In this phase, the TOCBUF is removed from the logical description and the global resources are used for this function.
- *Back-annotated Phase*—In this phase, use the NetGen option –gp to replace the port that was previously occupied by the TOCBUF in the RTL description of the design. A non-hierarchical netlist will have this port drive the fully wired GTS signal. For hierarchical designs, the GTS net is not wired across hierarchical modules. The VHDL global signal, X_GTS_GLOBAL_SIGNAL, is used to connect the top level port created by the -tp switch to the GTS signals of the hierarchical modules. Each hierarchical module contains an XTROCBUF cell which is driven by X_GTS_GLOBAL_SIGNAL. The output of the X_TOCBUF is connected to the GTS signal of the hierarchical module. Toggling the input to the top level port affects the entire design. The X_GTS_GLOBAL_SIGNAL global signal is defined in a package within the VHDL netlist. The package name is *design_name*_ROCTOC.

  If desired, each of the hierarchical modules can be written out as a separate netlist using the NetGen –mhf switch to create multiple hierarchical files. This helps in analyzing lower level modules individually. Every netlist contains the *design_name*_ROCTOC package definition with X_GTS_GLOBAL_SIGNAL signal. When lower modules are simulated, though X_TOCBUF is connected to X_GTS_GLOBAL_SIGNAL, there will be no event on it. Hence the X_TOCBUF creates a pulse similar to X_TOC and the GTS of the module is toggled. The TOC pulse width of the X_TOCBUF component can be controlled using the –tpw switch for NetGen.

  If all the modules, including the top level design modules are compiled and analyzed together, then the top level port created via the –tp switch would drive the X_GTS_GLOBAL_SIGNAL and all X_TOCBUF cells. In this situation, the X_TOCBUF component does not create a pulse, but rather behaves like a buffer passing on the value of the X_GTS_GLOBAL_SIGNAL. The TOC pulse width of the X_TOCBUF component can be controlled using the –tpw switch for NetGen.

## Using VHDL STARTBUF_VIRTEX, STARTBUF_VIRTEX2 or STARTBUF_SPARTAN2 Block

The STARTUP_VIRTEX, STARTUP_VIRTEX2 and STARTUP_SPARTAN2 blocks can be instantiated to identify the GTS signal for implementation if the global reset or tristate is connected to a chip pin. However, these cells cannot be simulated as there is no simulation model for them.

The VHDL STARTBUF_VIRTEX, STARTBUF_VIRTEX2 and STARTBUF_SPARTAN2 blocks can do a pre-NGDBuild UNISIM simulation of the GTS signal. You can also correctly back-annotate a GTS signal by instantiating a STARTUP_VIRTEX, STARTBUF_VIRTEX, STARTUP_SPARTAN2, or STARTBUF_SPARTAN2 symbol and correctly connect the GTSIN input signal of the component.

See the following table for Virtex™, Virtex-II™ and Spartan-II™ correspondence of pins between STARTBUF and STARTUP.

*Table 6-6:* **Virtex™/II/E and Spartan-II™ STARTBUF/STARTUP Pins**

| STARTBUF Pin Names | Connection Points | STARTUP Pin Names |
|---|---|---|
| GSRIN | Global Set / Reset Port of Design | GSR |
| GTSIN | Global Tristate Port of Design | GTS |
| CLKIN | Port of Internal Logic | CLK |
| GTSOUT | All Output Buffers Tristate Control | N / A |
| GSROUT | All Registers Asynchronous Set / Reset | N / A |

## STARTBUF_VIRTEX Model Example

The following is an example of the STARTBUF_VIRTEX model.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
library UNISIM;
use UNISIM.all;
entity EX_STARTBUF is
  port (
        CLOCK, ENABLE, RESET, STP : in std_logic;
        CUP, CDOWN : out std_logic_vector (3 downto 0)
        );
end EX_STARTBUF;

architecture A of EX_STARTBUF is
  signal GTS_sig : std_logic;
  signal COUNT_UP, COUNT_DOWN : std_logic_vector (3 downto 0);
  signal ZERO : std_ulogic := ‘0’;

component STARTBUF_VIRTEX
  port (
        GSRIN, GTSIN, CLKIN : in std_logic;
        GSROUT, GTSOUT : out std_logic
        );
end component;
```

```
begin
  U1 :STARTBUF_VIRTEX port map (
        GTSIN=>STP,
        GSRIN=>ZERO,
        CLKIN=>ZERO
        GTSOUT=>GTS_sig
        );
UP_COUNTER : process (CLOCK, ENABLE, RESET)
begin
  if (RESET = ’1’) then
      COUNT_UP <= "0000";
  elsif (CLOCK’event AND CLOCK = ’1’) then
      if (ENABLE = ’1’) then
          COUNT_UP <= COUNT_UP + "0001";
      end if;
  end if;
end process UP_COUNTER;
DOWN_COUNTER : process (CLOCK, ENABLE, RESET, COUNT_DOWN)
begin
  if (RESET = ’1’ OR COUNT_DOWN = "0101") then
      COUNT_DOWN <= "1111";
  elsif (CLOCK’event AND CLOCK = ’1’) then
      if (ENABLE = ’1’) then
          COUNT_DOWN <= COUNT_DOWN - "0001";
      end if;
  end if;
end process DOWN_COUNTER;
CUP <= COUNT_UP when (GTS_sig=’0’ AND COUNT_UP /= "0000") else "ZZZZ";
CDOWN <= COUNT_DOWN when (GTS_sig = ’0’) else "ZZZZ";
end A;
```

## Simulating Special Components in VHDL

The following section provides a description and examples of using special components, such as the Block SelectRAM™ for Virtex™.

### Simulating CORE Generator™ Components in VHDL

For CORE Generator™ model simulation flows, see the *CORE Generator Guide*.

### Boundary Scan and Readback

The Boundary Scan and Readback circuitry cannot be simulated at this time.

### Generics Support Considerations

Some synthesis tools do not support the use of generics to attach attributes. If your synthesis tool does not support generics, use the special meta comment to make the code visible only to the simulator. Place the meta comments immediately before and after the generic declarations and mappings as follows:

```
-- synthesis translate_off
   generic (INIT: std_logic_vector(1 downto 0) := “10”);
-- synthesis translate_off
```

The attributes can then be passed to the implementation tools by defining them in the UCF file. Alternatively, the synthesis tool may support a mechanism to pass these attributes directly from the VHDL file without using the generics. Please see your synthesis tool documentation for specific details on attribute passing for your synthesis tool.

## Differential I/O (LVDS, LVPECL)

When targeting a Virtex-E™ or Spartan-IIE™ device, the inputs of the differential pair are currently modeled with only the positive side, whereas the outputs have both pairs, positive and negative. For details, please refer to Xilinx® Answer Record 8187 on the Xilinx® support web sit at http://support.xilinx.com for more details. This is not an issue for the Virtex™-II/II Pro/II Pro X or Spartan-3™ architecture as the differential buffers for Virtex-II™ and later architectures have been updated to accept both the positive and negative inputs.

The following is an example of an instantiated differential I/O in a Virtex-E™ or Spartan-IIE™ design.

```
entity lvds_ex is
port (
       data: in std_logic;
       data_op: out std_logic;
       data_on: out std_logic
       );
end entity lvds_ex;
architecture lvds_arch of lvds_ex is
  signal data_n_int : std_logic;
  component OBUF_LVDS port (
       I : in std_logic;
       O : out std_logic
       );
  end component;
  component IBUF_LVDS port (
       I : in std_logic;
       O : out std_logic
       );
  end component;
begin
--Input side
  I0: IBUF_LVDS port map (I => data), O =>data_int);
--Output side
  OP0: OBUF_LVDS port map (I => data_int, O => data_op);
  data_n_int = not(data_int);
  ON0: OBUF_LVDS port map (I => data_n_int, O => data_on);
end arch_lvds_ex;
```

## Simulating a LUT

The LUT (look-up table) component is initialized for simulation by a generic mapping to the INIT attribute. If the synthesis tool being used can accept generics in order to pass attributes, then a generic specification is all that is needed to specify the INIT value. If the synthesis tool cannot pass attributes via generics, then the generic and generic map portions of the code must be omitted for synthesis by the use of translate_off and translate_on synthesis directives. The INIT values must be passed using attribute notation.

The following is an example in which a LUT is initialized. This code written with the assumption that the synthesis tool can understand and pass the INIT attribute using the generic notation.

```
entity lut_ex is
  port (
       LUT1_IN, LUT2_IN : in std_logic_vector(1 downto 0);
       LUT1_OUT, LUT2_OUT : out std_logic_vector(1 downto 0)
       );
end entity lut_ex;
architecture lut_arch of lut_ex is
  component LUT1
   generic (INIT: std_logic_vector(1 downto 0) := “10”);
   port (
       O : out std_logic;
       I0 : in std_logic
       );
  end component;
  component LUT2
   generic (INIT: std_logic_vector(3 downto 0) := “0000”);
   port (
       O : out std_logic;
       I0, I1: in std_logic
       );
  end component;
begin
-- LUT1 used as an inverter
  U0: LUT1 generic map (INIT => “01”)
  port map (O => LUT1_OUT(0), I0 => LUT1_IN(0));
-- LUT1 used as a buffer
  U1: LUT1 generic map (INIT => “10”)
  port map (O => LUT1_OUT(1), I0 => LUT1_IN(1));
--LUT2 used as a 2-input AND gate
  U2: LUT2 generic map (INIT => “1000”)
  port map (O => LUT2_OUT(0), I1 => LUT2_IN(1), I0 => LUT2_IN(0));
--LUT2 used as 2-input NAND gate
  3: LUT2 generic map (INIT => “0111”)
  port map (O => LUT2_OUT(1), I1 => (LUT2_IN(1), I0 => LUT2_IN(0));
end lut_arch;
```

***Note:*** Some synthesis tools may not support the use of a generic statements. See the “Generics Support Considerations” section for details.

## Simulating Virtex™ Block SelectRAM™

By default, the Virtex™ Block SelectRAM™ comes up initialized to zero in all data locations starting at time zero. If you want to initialize the RAM or ROM contents to anything other than zero, you can do this by applying the appropriate INIT_xx generics in the VHDL code.

The following is an example of using generic mapping to apply an initial value to a Block SelectRAM™. This code was written with the assumption that the synthesis tool can understand and pass the INIT attribute using the generic notation.

```
LIBRARY ieee;
use IEEE.std_logic_1164.all;
Library UNISIM;
use UNISIM.vcomponents.all;
entity ex_blkram is
  port(
        CLK, EN, RST, WE : in std_logic;
        ADDR : in std_logic_vector(9 downto 0);
        DI : in std_logic_vector(3 downto 0);
        DORAMB4_S4 : out std_logic_vector(3 downto 0)
        );
end;

architecture struct of ex_blkram is
component RAMB4_S4
    generic (
        INIT_00, INIT_01, INIT_02 : bit_vector;
        INIT_03, INIT_04, INIT_05 : bit_vector;
        INIT_06, INIT_07, INIT_08 : bit_vector;
        INIT_09, INIT_0A, INIT_0B : bit_vector;
        INIT_0C, INIT_0D, INIT_0E : bit_vector;
        INIT_0F : bit_vector
        );
    port (
        DI : in STD_LOGIC_VECTOR (3 downto 0);
        EN : in STD_ULOGIC;
        WE : in STD_ULOGIC;
        RST : in STD_ULOGIC;
        CLK : in STD_ULOGIC;
        ADDR : in STD_LOGIC_VECTOR (9 downto 0);
        DO : out STD_LOGIC_VECTOR (3 downto 0)
        );
end component;
begin
  INST_RAMB4_S4 : RAMB4_S4
  generic map (
        INIT_00 => X"new_hex_value",
        INIT_01 => X"new_hex_value",
        INIT_02 => X"new_hex_value",
        INIT_03 => X"new_hex_value",
        INIT_04 => X"new_hex_value",
        INIT_05 => X"new_hex_value",
        INIT_06 => X"new_hex_value",
        INIT_07 => X"new_hex_value",
        INIT_08 => X"new_hex_value",
        INIT_09 => X"new_hex_value",
        INIT_0A => X"new_hex_value",
        INIT_0B => X"new_hex_value",
        INIT_0C => X"new_hex_value",
        INIT_0D => X"new_hex_value",
        INIT_0E => X"new_hex_value",
        INIT_0F => X"new_hex_value"
        );
   port map (
        DI => DI,
        EN => EN,
        WE => WE,
        RST => RST,
        CLK => CLK,
        ADDR => ADDR,
        DO => DORAMB4_S4
        );
end struct;
```

***Note:*** Some synthesis tools may not support the use of a generic statements. See the "Generics Support Considerations" section for details.

## Simulating the Virtex™ Clock DLL

When functionally simulating the Virtex™ Clock DLL, generic maps are used to specify the CLKDV_DIVIDE and DUTY_CYCLE_CORRECTION values. By default, the CLKDV_DIVIDE is set to 2 and DUTY_CYCLE_CORRECTION is set to **TRUE**. The following example sets the CLKDV_DIVIDE to 4, and sets the DUTY_CYCLE_CORRECTION to **FALSE**.

This code was written with the assumption that the synthesis tool can understand and pass the INIT attribute using generic notation.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
Library UNISIM;
use UNISIM.vcomponents.all;
entity clkdlls is port(
        CLK_LF, RST_LF : in std_logic;
        CLK90_LF, CLK180_LF : out std_logic;
        CLK270_LF, CLK2X_LF : out std_logic;
        CLKDV_LF, LOCKED_LF : out std_logic;
        LFCount : out std_logic_vector(3 downto 0)
        );
end;
architecture struct of clkdlls is
  component CLKDLL
    generic (
        FACTORY_JF : bit_vector := X"00";
        STARTUP_WAIT : boolean := false;
        DUTY_CYCLE_CORRECTION : boolean := TRUE;
        CLKDV_DIVIDE : real := 2.0);
    port (
        CLKIN  : in std_logic;
        CLKFB  : in std_logic;
        RST    : in std_logic;
        CLK0   : out std_logic;
        CLK90  : out std_logic;
        CLK180 : out std_logic;
        CLK270 : out std_logic;
        CLK2X  : out std_logic;
        CLKDV  : out std_logic;
        LOCKED : out std_logic);
  end component;
```

```
    component IBUFG
      port (
          I : in std_logic;
          O : out std_logic
          );
    end component;
    component BUFG
      port (
          I : in std_logic;
          O : out std_logic
          );
    end component : integer range 0 to 15 := 0;
    signal sigCLK_LF, sigCLK0_LF, sigCLKFB_LF, CLK0_LF : std_logic;
    signal sigLFCount : std_logic_vector (3 downto 0);

    begin
      INST_IBUFGLF : IBUFG port map (I => CLK_LF, O => sigCLK_LF);
      INST_BUFGLF : BUFG port map (I => sigCLK0_LF, O => sigCLKFB_LF);
      INST_CLKDLL : CLKDLL
      generic map (
          DUTY_CYCLE_CORRECTION => FALSE,
          CLKDV_DIVIDE => 4.0
          )
      port map (
          CLKIN  => sigCLK_LF,
          CLKFB  => sigCLKFB_LF,
          RST    => RST_LF,
          CLK0   => sigCLK0_LF,
          CLK90  => CLK90_LF,
          CLK180 => CLK180_LF,
          CLK270 => CLK270_LF,
          CLK2X  => CLK2X_LF,
          CLKDV  => CLKDV_LF,
          LOCKED => LOCKED_LF
          );
      CLK0_LF <= sigCLK0_LF;

    procCLKDLLCount: process (CLK0_LF)
    begin
      if (CLK0_LF'event and CLK0_LF = '1') then
          sigLFCount <= sigLFCount + "0001";
      end if;
      LFCount <= sigLFCount;
    end process;
  end struct;
```

***Note:*** Some synthesis tools may not support the use of a generic statements. See the "Generics Support Considerations" section for details.

## Simulating the Virtex™-II/II Pro/II Pro X/Spartan-3™ DCM

The Virtex-II™ / Virtex-II Pro™ / Virtex-II Pro X™ / Spartan-3™ DCM is a super set of the Virtex™ CLKDLL. It provides more clock options, including fine phase shifting and digital clock synthesis. The DCM attributes, like all UNISIM components, are specified via generics for simulation purposes and most synthesis tools can read in the generics for passing to the implementation tools.

Following is an example of the DCM instantiation. Note the component declaration of the DCM, as the parameters are defined in the "generic" section of the component declaration. In order to use some of the DCM features, these generic values must be modified.

```
library ieee;
use ieee.std_logic_1164.all;
library unisim;
use unisim.vcomponents.all;

entity clock_gen is
  port (
       clkin, rst_dll: in std_logic;
       clk, clk_not, locked : out std_logic;
       psen, psclk, psincdec : in std_logic;
       psdone : out std_logic
       );
end clock_gen;

architecture structural of clock_gen is
  signal clk_ibufg, clk_dcm, clk_dcm_not : std_logic;
  signal clk0_bufg, clk180_bufg : std_logic;
  signal GND : std_logic;

  component IBUFG
    port (
       I : in std_logic;
       O : out std_logic
       );
  end component;
  component BUFG
    port (
       I : in std_logic;
       O : out std_logic
       );
  end component;
  component DCM
    generic (
       DFS_FREQUENCY_MODE : string := "LOW";
       DLL_FREQUENCY_MODE : string := "LOW";
       DUTY_CYCLE_CORRECTION : boolean := TRUE;
       CLKIN_DIVIDE_BY_2 : boolean := FALSE;
       CLK_FEEDBACK : string := "1X";
       CLKOUT_PHASE_SHIFT : string := "NONE";
       FACTORY_JF : bit_vector := X"00";
       STARTUP_WAIT : boolean := FALSE;
       DSS_MODE : string := "NONE";
       PHASE_SHIFT  : integer := 0;
       CLKFX_MULTIPLY : integer  := 4;
       CLKFX_DIVIDE : integer  := 1;
       CLKDV_DIVIDE : real := 2.0;
       DESKEW_ADJUST : string := "SYSTEM_SYNCHRONOUS"
       );
   port (
       CLKIN    : in std_ulogic;
       CLKFB    : in std_ulogic;
       DSSEN    : in std_ulogic;
       PSINCDEC : in std_ulogic;
       PSEN     : in std_ulogic;
       PSCLK    : in std_ulogic;
       RST      : in std_ulogic;
       CLK0     : out std_ulogic;
       CLK90    : out std_ulogic;
       CLK180   : out std_ulogic;
       CLK270   : out std_ulogic;
       CLK2X    : out std_ulogic;
       CLK2X180 : out std_ulogic;
       CLKDV    : out std_ulogic;
       CLKFX    : out std_logic;
       CLKFX180 : out std_logic;
       LOCKED   : out std_ulogic;
       PSDONE   : out std_ulogic;
       STATUS   : out std_logic_vector(7 downto 0)
       );
  end component;

  begin
   GND <= ’0’;
   U1 : IBUFG port map (
       I => clkin,
       O => clk_ibufg);
   U2 : DCM
     generic map (
       DFS_FREQUENCY_MODE => "LOW",
       DLL_FREQUENCY_MODE => "LOW",
       DUTY_CYCLE_CORRECTION => TRUE,
       CLKIN_DIVIDE_BY_2 => FALSE,
       CLK_FEEDBACK => "1X",
       CLKOUT_PHASE_SHIFT => "VARIABLE",
       FACTORY_JF => X"00",
       STARTUP_WAIT => FALSE,
       DSS_MODE => "NONE",
       PHASE_SHIFT => 0,
       CLKFX_MULTIPLY => 4,
       CLKFX_DIVIDE => 1,
       CLKDV_DIVIDE => 2.0,
       DESKEW_ADJUST => "SYSTEM_SYNCHRONOUS")
     port map (
       CLKIN => clk_ibufg,
       CLKFB => clk0_bufg,
       DSSEN => ’0’,
       PSINCDEC => psincdec,
       PSEN => psen,
       PSCLK => psclk,
       PSDONE => psdone,
       RST => rst_dll,
       CLK0 => clk_dcm,
       CLKDV => open,
       CLKFX => open,
       CLK180 => clk_dcm_not,
       LOCKED => locked
       );
  U3 : BUFG port map (
       I => clk_dcm,
       O => clk0_bufg
       );
  U4 : BUFG port map (
       I => clk_dcm_not,
       O => clk180_bufg
       );
  clk <= clk0_bufg;
  clk_not <= clk180_bufg;
end structural;
```

***Note:*** Some synthesis tools may not support the use of a generic statements. See the "Generics Support Considerations" section for details.

## Simulating SRLs

Most synthesis tools infer the SRL16 from behavioral VHDL. For these designs, no special simulation steps are needed for the SRLs. However, when the SRL component is instantiated, the INIT attribute can be used to initialize the contents of the component. Also, to use the select lines of the SRL component, instantiation is generally necessary. Refer to "Implementing Shift Registers (Virtex™/E/II/II Pro/II Pro X and Spartan™-II/3)" for more details on inferring SRLs correctly in the design.

Following is an example of passing the INIT attribute to the SRL for functional simulation:

***Note:*** If the synthesis tool being used can accept generics to pass attributes, then a generic specification is all that is needed to specify the INIT value to the implementation tools. If the synthesis tool cannot pass attributes via generics, then the generic and generic map portions of the code must be omitted for synthesis by the use of translate_off and translate_on synthesis directives and the INIT values must be passed using the attribute notation.

```
    entity design is
    -- port list goes here
    end entity design;
    architecture toplevel of designs
      component SRL16
      generic (INIT : BIT_VECTOR := X”0000”);
      port (
           D   : in STD_ULOGIC;
           CLK : in STD_ULOGIC;
           A0  : in STD_ULOGIC;
           A1  : in STD_ULOGIC;
           A2  : in STD_ULOGIC;
           A3  : in STD_ULOGIC;
           Q   : out STD_ULOGIC
           );
      end component;
    -- signal declarations go here
      begin
        U0 : SRL16 generic map (INIT => X”1100”);
        port map (
           CLK => CLK,
        -- rest of port maps
           );
    end toplevel;
```

In the example above, the INIT attribute is passed down to the simulation model through the generic map.

# Simulating Verilog

## Defining Global Signals in Verilog

The global set/reset and global tristate signals are defined in the `$XILINX/verilog/src/glbl.v` module. The VHDL UNISIM library contains the ROC, ROCBUF, TOC, TOCBUF, and STARTBUF cells to assist in VITAL VHDL simulation of the global set/reset and tristate signals. However, Verilog allows a global signal to be modeled as a wire in a global module, and thus, does not contain these cells.

## Using the glbl.v Module

The `glbl.v` module connects the global signals to the design, which is why it is necessary to compile this module with the other design files and load it along with the *`design`*`.v` file and the *`testfixture`*`.v` file for simulation.

***Note:*** In previous releases of the software, the `glbl.v` file only declared the global signals. In the 6.1i release, code has been added to the `glbl.v` file that automatically pulses GSR for 100 ns at the beginning of simulation.

The following is the definition of the `glbl.v` file.

```
‘timescale 1 ps / 1 ps

module glbl ();

  parameter ROC_WIDTH = 100000;
  parameter TOC_WIDTH = 0;

  wire GSR;
  wire GTS;
  wire PRLD;
  reg GSR_int;
  reg GTS_int;
  reg PRLD_int;

  assign (weak1, weak0) GSR = GSR_int;
  assign (weak1, weak0) GTS = GTS_int;
  assign (weak1, weak0) PRLD = PRLD_int;

  initial begin
    GSR_int = 1’b1;
    PRLD_int = 1’b1;
    #(ROC_WIDTH)
    GSR_int = 1’b0;
    PRLD_int = 1’b0;
  end

  initial begin
    GTS_int = 1’b1;
    #(TOC_WIDTH)
    GTS_int = 1’b0;
  end
endmodule
```

## Defining GSR/GTS in a Test Bench

In most cases, GSR and GTS need not be defined in the test bench. The `glbl.v` file declares the global GSR and GTS signals and automatically pulses GSR for 100 ns. This is all that is necessary for back-end simulations and is generally all that is necessary for functional simulations. If GSR or GTS needs to be emulated in the functional simulation, then it is necessary to add GSR and GTS to the test bench and connect them to the user defined global signals. This is discussed in more detail in the following section.

***Note:*** The terms “test bench” and “test fixture” are used synonymously throughout this manual.

## Emulating the Global GSR in a Verilog Functional Simulation

Many HDL designs that target Xilinx® FPGAs have a user reset that initializes all registers in the design during the functional simulation. For these designs, it is not necessary to emulate the GSR pulse that occurs after the device is configured. If the design contains registers that are not connected to a user reset, the GSR pulse can be emulated to ensure that the functional simulation matches the timing simulation.

In the design code, declare a GSR as a Verilog wire. The GSR is not specified in the port list for the module. Describe the GSR to reset or set every inferred register or latch in your design. GSR need not be connected to any instantiated registers or latches, as the UNISIM models for these components connect to GSR directly. This is shown in the following example.

```
module my_counter (CLK, D, Q, COUT);
input CLK, D;
output Q;
output [3:0] COUT;

wire GSR;
reg [3:0] COUT;

always @(posedge GSR or posedge CLK)
  begin
    if (GSR == 1’b1)
        COUT = 4’h0;
    else
        COUT = COUT + 1’b1;
  end
// GSR is modeled as a wire within a global module.
// So, CLR does not need to be connected to GSR and
// the flop will still be reset with GSR.
FDCE U0 (.Q (Q), .D (D), .C (CLK), .CE (1'b1), .CLR (1’b0));
endmodule
```

Since GSR is declared as a floating wire and is not in the port list, the synthesis tool optimizes the GSR signal out of the design. GSR is replaced later by the implementation software for all post-implementation simulation netlists.

In the test fixture file, set `testbench.uut.GSR` to `glbl.GSR`. See the following example:

```
’timescale 1 ns / 1 ps
module testbench;
  reg CLK, D;
  wire Q;
  wire [3:0] COUT;
  assign testbench.uut.GSR = glbl.GSR;  // glbl.GSR asserted in glbl.v
  my_counter uut (.CLK (CLK), .D (D), .Q (Q), .COUT (COUT));
  initial begin
    $timeformat(-9,1,”ns”,12);
    $display(“\t   T C G D Q C”);
    $display(“\t   i L S     O”);
    $display(“\t   m K R     U”);
    $display(“\t   e         T”);
    $monitor(“%t %b %b %b %b %h”, $time, CLK, GSR, D, Q, COUT);
  end
  initial begin
    CLK = 0;
    forever #25 CLK = ~CLK;
  end
// Apply Design Stimulus here
  initial begin
    D = 1'b1;
    #100 D = 1'b0;
    #200 D = 1'b1;
    #100 $finish;
  end
endmodule
```

# Simulating Special Components in Verilog

The following section provides descriptions and examples of simulating special components for Virtex™.

## Boundary Scan and Readback

The Boundary Scan and Readback circuitry cannot be simulated at this time.

## Defparam Support Considerations

Some synthesis tools do not support the use of defparams to attach attributes. If your synthesis tool does not support defparams, use of the special meta comment to make the code visible only to the simulator. Place the meta comments immediately before and after the defparam declarations and mappings as follows:

```
// synthesis translate_off
   defparam U0.INIT = 2’b01;
// synthesis translate_off
```

The attributes can then be passed to the implementation tools by defining them in the UCF file. Alternatively, the synthesis tool may support a mechanism to pass these attributes directly from the Verilog file without using the generics. Please see your synthesis tool documentation for specific details on attribute passing for your synthesis tool.

## Differential I/O (LVDS, LVPECL)

For Virtex-E™ and Spartan-IIE™ families, the inputs of the differential pair are currently modeled with only the positive side, whereas the outputs have both pairs, positive and negative. For details, please see Answer Record 8187 on the Xilinx® support website at http://support.xilinx.com.

This is not an issue for the Virtex™-II / II Pro / II Pro X or Spartan-3™ architecture because the differential buffers for Virtex-II™ and later architectures have been updated to accept both the positive and negative inputs.

The following is an example of an instantiated differential I/O in a Virtex-E™ or Spartan-IIE™ design.

```
module lvds_ex (data, data_op, data_on);
  input data;
  output data_op, data_on;

// Input side
  IBUF_LVDS I0 (.I (data), .O (data_int));

// Output side
  OBUF_LVDS OP0 (.I (data_int), .O (data_op));
  wire data_n_int = ~data_int;
  OBUF_LVDS ON0 (.I (data_n_int), .O (data_on));

endmodule
```

## LUT

For simulation, the INIT attribute passed by the defparam statement is used to initialize contents of the LUT for functional simulation.

The following is an example of the defparam statement being used to initialize the contents of a LUT.

```
module lut_ex (LUT1_OUT, LUT1_IN);
  input  [1:0] LUT1_IN;
  output [1:0] LUT1_OUT;

// LUT1 used as an inverter
  LUT1 U0 (.O (LUT1_OUT[0]), .I0 (LUT1_IN[0]));
  defparam U0.INIT = 2'b01;

// LUT1 used as a buffer
  LUT1 U1 (.O (LUT1_OUT[1]), .I0 (LUT1_IN[1]));
  defparam U1.INIT = 2'b10;

endmodule
```

***Note:*** Some synthesis tools may not support the use of a defparam statements. See the "Defparam Support Considerations" section for details.

## SRL16

For inferred SRL16s, no attributes need to be passed to the simulator. However, if the SRL16 component is being instantiated, and if non-zero contents are desired for initialization, the INIT attribute passed by the defparam statement is used to initialize contents of the SRL16.

The following is an example of the defparam statement being used to initialize the contents of an SRL16.

```
module srl16_ex (CLK, DIN, QOUT);
  input CLK, DIN;
  output QOUT;

// Static length - 16-bit SRL
  SRL16 U0 (.D (DIN), .Q (QOUT), .CLK (CLK), .A0 (1’b1), .A1 (1’b1),
   .A2 (1’b1), .A3 (1’b1));
  defparam U0.INIT = 16’hAAAA;
endmodule
```

***Note:*** Some synthesis tools may not support the use of a defparam statements. See the “Defparam Support Considerations” section for details.

## Block RAM

For simulation, the INIT_0x attributes passed by the defparam statement are used to initialize contents of the block ram.

```
MODULE bram512x4 (CLK, DATA_BUSA, ADDRA, WEA, DATA_BUSB, ADDRB, WEB);
  input [9:0] ADDRA, ADDRB;
  input CLK, WEA, WEB;
  inout [3:0] DATA_BUSA, DATA_BUSB;

  wire [3:0] DOA, DOB;

  assign DATA_BUSA = !WEA ? DOA : 4’hz;
  assign DATA_BUSB = !WEB ? DOB : 4’hz;

  RAMB4_S4_S4 U0 (.DOA (DOA), .DOB (DOB), .ADDRA (ADDRA),
   .DIA (DATA_BUSA), .ENA (1’b1), .CLKA (CLK), .WEA (WEA),
   .RSTA (1’b0), .ADDRB (ADDRB), .DIB (DATA_BUSB),
   .ENB (1’b1), .CLKB (CLK), .WEB (WEB), .RSTB (1’b0));
  defparam
    U0.INIT_00 = 256'hnew_hex_value,
    U0.INIT_01 = 256'hnew_hex_value,
    U0.INIT_02 = 256'hnew_hex_value,
    U0.INIT_03 = 256'hnew_hex_value,
    U0.INIT_04 = 256'hnew_hex_value,
    U0.INIT_05 = 256'hnew_hex_value,
    U0.INIT_06 = 256'hnew_hex_value,
    U0.INIT_07 = 256'hnew_hex_value,
    U0.INIT_08 = 256'hnew_hex_value,
    U0.INIT_09 = 256'hnew_hex_value,
    U0.INIT_0A = 256'hnew_hex_value,
    U0.INIT_0B = 256'hnew_hex_value,
    U0.INIT_0C = 256'hnew_hex_value,
    U0.INIT_0D = 256'hnew_hex_value,
    U0.INIT_0E = 256'hnew_hex_value,
    U0.INIT_0F = 256'hnew_hex_value;

endmodule
```

***Note:*** Some synthesis tools may not support the use of a defparam statements. See the “Defparam Support Considerations” section for details.

Another method for passing the INIT_0x attributes is through the use of a UCF file. For example, the following statement defines the initialization string for the code example above.

```
INST U0 INIT_00 = 5555aaaa5555aaaa5555aaaa5555aaaa5555aaaa;
INST U0 INIT_01 = 5555aaaa5555aaaa5555aaaa5555aaaa5555aaaa;
```

The value of the INIT_0x string is a hexadecimal number that defines the initialization string for implementation however, it will not be reflected for behavioral simulation.

When using Distributed RAM or Block RAM in dual-port mode, special care must be taken to avoid memory collisions. See the "Collision Checking" section for a general description of collision checking.

## CLKDLL

The duty cycle of the CLK0 output is 50% unless the DUTY_CYCLE_CORRECTION attribute is set to FALSE, in which case the duty cycle is the same as that of the CLKIN.

The frequency of CLKDV is determined by the value assigned to the CLKDV_DIVIDE attribute. The default is 2.

The STARTUP_WAIT is not implemented in the model however; it can be passed via a defparam similar to CLKDV_DIVIDE and DUTY_CYCLE_CORRECTION.

```
module clkdll_ex (CLKIN_P, RST_P, CLK0_P, CLK90_P, CLK180_P,
    CLK270_P,CLK2X_CLKDV_P, LOCKED_P);
  input CLKIN_P, RST_P;
  output CLK0_P, CLK90_P, CLK180_P, CLK270_P, CLK2X_P;
  output CLKDV_P;
// Active high indication that DLL is
// LOCKED to CLKIN
  output LOCKED_P;
  wire CLKIN, CLK0;

// Input buffer on the clock
  IBUFG U0 (.I (CLKIN_P), .O (CLKIN));

// GLOBAL CLOCK BUFFER on the
// delay compensated output
  BUFG U2 (.I (CLK0), .O (CLK0_P));

// Instantiate the DLL primitive cell
  CLKDLL DLL0 (.CLKIN (CLKIN), .CLKFB(CLK0_P), .RST (RST_P),
    .CLK0 (CLK0), .CLK90 (CLK90_P), .CLK180 (CLK180_P),
    .CLK270 (CLK270_P), .CLK2X (CLK2X_P), .CLKDV (CLKDV_P),
    .LOCKED (LOCKED_P));
// CLK0 divided by
// 1.5 2.0 2.5 3.0 4.0 5.0 8.0 or 16.0
  defparam DLL0.CLKDV_DIVIDE = 4.0;
  defparam DLL0.DUTY_CYCLE_CORRECTION = "FALSE";
endmodule
```

***Note:*** Some synthesis tools may not support the use of a defparam statements. See the "Defparam Support Considerations" section for details.

Another method for passing the CLKDLL attributes is through the use of a UCF file. For example, the following statement defines the initialization string for the code example above. Note that passing attributes via the UCF file will not be reflected in behavioral simulation. Care must be taken to ensure that there is not a miscorelation between behavioral simulation and implementation by ensuring that the defparam and UCF attributes are the same.

```
INST DLL0 CLKDV_DIVIDE = 4;
INST DLL0 DUTY_CYCLE_CORRECTION = FALSE;
```

## DCM

The DCM (Digital Clock Management) component, available in Virtex-II™, Virtex-II Pro™, Virtex-II Pro X™ and Spartan-3™ is an enhancement over the Virtex-E™ CLKDLL. The following example shows how to pass the attributes to the DCM component using the defparam statement in Verilog.

```
module DCM_TEST(clock_in, clock_out, clock_with_ps_out, reset);
  input clock_in;
  output clock_out;
  output clock_with_ps_out;
  output reset;

  wire low, high, dcm0_locked, reset, clk0;

  assign low = 1’b0;
  assign high = 1’b1;
  assign reset = !dcm0_locked;

  IBUFG CLOCK_IN ( .I(clock_in), .O(clock) );

  DCM DCM0 (
    .CLKFB(clock_out), .CLKIN(clock), .DSSEN(low), .PSCLK(low),
    .PSEN(low), .PSINCDEC(low), .RST(low), .CLK0(clk0), .CLK90(),
    .CLK180(), .CLK270(), .CLK2X(), .CLK2X180(), .CLKDV(), .CLKFX(),
    .CLKFX180(), .LOCKED(dcm0_locked), .PSDONE(), .STATUS() );
  defparam DCM0.DLL_FREQUENCY_MODE = "LOW";
  defparam DCM0.DUTY_CYCLE_CORRECTION = "TRUE";
  defparam DCM0.STARTUP_WAIT = "TRUE";
  defparam DCM0.DFS_FREQUENCY_MODE = "LOW";
  defparam DCM0.CLKFX_DIVIDE = 1;
  defparam DCM0.CLKFX_MULTIPLY = 1;
  defparam DCM0.CLK_FEEDBACK = "1X";
  defparam DCM0.CLKOUT_PHASE_SHIFT = "NONE";
  defparam DCM0.PHASE_SHIFT = "0";
  defparam DCM0.CLK_FEEDBACK = "1X";
  defparam DCM0.CLKIN_DIVIDE_BY_2 = FALSE;
  defparam DCM0.CLKIN_PERIOD = 0.0;
  defparam DCM0.DESKEW_ADJUST = "SYSTEM_SYNCHRONOUS";
  defparam DCM0.DFS_FREQUENCY = "LOW";
  defparam DCM0.DLL_FREQUENCY = "LOW";
  BUFG CLK_BUF0( .O(clock_out), .I(clk0) );
endmodule // DCM_TEST
```

***Note:*** Some synthesis tools may not support the use of a defparam statements. See the “Defparam Support Considerations” section for details.

### Simulation CORE Generator™ Components

The simulation flow for CORE Generator™ models is described in the *CORE Generator Guide*.

## Design Hierarchy and Simulation

Most FPGA designs are partitioned into levels of hierarchy for many advantageous reasons. A few of the reasons are that hierarchy makes the design easier to read, easier to re-use, allows partitioning for a multi-engineer team, and improves verification. To improve design utilization and performance, many times the synthesis tool or the Xilinx® implementation tools flatten or modifie the design hierarchy. After this flattening and restructuring of the design hierarchy in synthesis and implementation, many times it becomes impossible to reconstruct this hierarchy and thus, much of the advantage of using the original design hierarchy in RTL verification is lost in back-end verification. In an effort to improve visibility of the design for back-end simulation, the Xilinx® design flow allows for retention of the original design hierarchy with this described methodology.

To allow preservation of the design hierarchy through the implementation process with little or no degradation in performance or increase in design resources, stricter design rules should be followed and design hierarchy should be carefully selected so that optimization is not necessary across the design hierarchy.

Some good design practices to follow are:

- Register all outputs exiting a preserved entity or module.
- Do not allow critical timing paths to span multiple entities or modules.
- Keep related or possibly shared logic in the same entity or module.
- Place all logic that is to be placed / merged into the I / O (IOB registers, three state buffers, instantiated I / O buffers, etc.) in the top-level module or entity for the design. This includes double-data rate registers used in the I / O.
- Manually duplicate high-fanout registers at hierarchy boundaries if improved timing is necessary.

Generally, it is good practice to follow the guidelines in the *FPGA Reuse Field Guide*. You can find this manual in PDF format on the Xilinx® website at http://www.xilinx.com/ipcenter/designreuse/index.htm.

To maintain the entire hierarchy or specified parts of the hierarchy during synthesis, the synthesis tool must first be instructed to preserve hierarchy for all levels or each selected level of hierarchy. This may be done with a global switch, compiler directive in the source files, or a synthesis command. Consult your synthesis tool documentation for details on how to retain hierarchy. After taking the necessary steps to preserve hierarchy, and properly synthesizing the design, a hierarchical implementation file (EDIF or NGC) will be created by the synthesis tool that retains the hierarchy.

Before implementing the design with the Xilinx® software, place a KEEP_HIERARCHY constraint on each instance in the design in which the hierarchy is to be preserved. This tells the Xilinx® software exactly which parts of the design should not be flattened or modified to maintain proper hierarchy boundaries. This constraint may be passed in the source code as an attribute, as an instance constraint in the NCF or UCF file, or may be automatically generated by the synthesis tool. See your synthesis vendor documentation to see how your synthesis tool handles this. More information on the KEEP_HIERARCHY constraint can be found in the *Constraints Guide.*

Alternatively, if the design was compiled using a bottom-up methodology where individual implementation files (EDIF or NGC) were created for each level of design hierarchy, the KEEP_HIERARCHY constraint may be automatically generated. A KEEP_HIERARCHY constraint is generated for each separate design file passed to the Xilinx® software by the use of a switch during input netlist translation. During the NGDBuild netlist translation stage, if the –insert_keep_hierarchy switch is enabled, the hierarchy for each individual input file for the design is preserved during implementation.

After the design is mapped, placed, and routed, run NetGen with the resulting NGM file from Map during delay annotation to properly back-annotate the hierarchy of the design.

```
netgen –sim -ofmt {vhdl|verilog} –ngm design_name_map.ngm
       design_name.ncd netlist_name
```

This is the default way NetGen is run when using ISE or XFLOW to generate the simulation files. It is only necessary to know this if you plan to execute NetGen outside of ISE or XFLOW, or if you have modified the default options in ISE or XFLOW. When you run NetGen using the NGM file, all hierarchy that was specified to KEEP_HIERARCHY will be reconstructed in the resulting VHDL or Verilog netlist.

NetGen has the ability to write out a separate netlist file and SDF file for each level of preserved hierarchy. This capability allows for full timing simulation of individual portions of the design, thus allowing for greater test bench re-use, team-based verification methods and potentially cutting down overall verification times. To have NetGen produce individual files for each KEEP_HIERARCHY instance in the design, use the –mhf switch. You can use this switch in conjunction with the –dir switch to have all associated files placed in a separate directory:

```
netgen -sim -ofmt {vhdl|verilog} -mhf -ngm design_name_map.ngm -dir
       directory_name design_name.ncd
```

When NetGen is invoked with the –mhf switch, it will also produce a text file called `design_mhf_info.txt`. This file will list all produced module/entity names, their associated instance names, SDF files and sub modules. This file can be useful for determining proper simulation compile order, SDF annotation options, and other information when using one or more of these files for simulation.

The following is an example of an `mhf_info.txt` file for a Verilog produced netlist:

```
// Xilinx design hierarchy information file produced
//    by netgen (build+G-19+0)
// The information in this file is useful for
//   - Design hierarchy relationship between modules
//   - Bottom up compilation order (VHDL simulation)
//   - SDF file annotation (VHDL simulation)
//
//  Design Name : design_top
//
//  Module      : The name of the hierarchical design module.
//  Instance    : The instance name used in the parent module.
//  Design File : The name of the file that contains the module.
//  SDF File    : The SDF file associated with the module.
//  SubModule   : The sub module(s) contained within a given module.
//     Module, Instance : The sub module and instance names.

  Module      : submodule
  Instance    : submodule_inst
  Design File : submodule_sim.v
  SDF File    : submodule_sim.sdf
  SubModule   : NONE
```

```
Module      : design_top
Design File : design_top_timesim.v
SDF File    : design_top_timesim.sdf
SubModule   : submodule
     Module : submodule, Instance : submodule_inst
```

***Note:*** Hierarchy created by generate statements may not match the original simulation due to naming differences between the simulator and synthesis engines for generated instances.

# RTL Simulation Using Xilinx® Libraries

Since Xilinx® simulation libraries are VHDL-93 and Verilog-2001 compliant, they can be simulated using any simulator that supports these language standards. However, certain delay and modelling information is built into the libraries, which is required to correctly simulate the Xilinx® hardware devices.

Xilinx® recommends not changing data signals at clock edges, even for functional simulation. The simulators add a unit delay between the signals that change at the same simulator time. If the data changes at the same time as a clock, it is possible that the data input will be scheduled by the simulator to occur after the clock edge. Thus, the data will not go through until the next clock edge, although it is possible that the intent was to have the data get clocked in before the first clock edge. To avoid any such unintended simulation results, Xilinx® recommends not switching data signals and clock signals simultaneously.

The UNISIM dual-port Block RAM models have a built-in collision checking function that monitors reads and writes to each port, and reports violations if data is improperly handled for the RAM. While this is reported similarly to a timing violation, in reality, it is warning of a potential functionality issue for the design. If you receive any collision warnings from the UNISIM dual-port Block RAM models during functional simulation, you should investigate the warning to ensure it will not have any negative impact on the end design functionality. Generally, this is best handled either by trying to re-code to avoid such errors or by ensuring that the data written or received from the RAM will be discarded as the actual value cannot be determined and so should not be used.

# Timing Simulation

In back annotated (timing) simulation, the introduction of delays can cause the behavior to be different from what is expected. Most problems are caused due to timing violations in the design, and are reported by the simulator. However, there are a few other situations that can occur.

## Glitches in your Design

When a glitch (small pulse) occurs in an FPGA circuit or any integrated circuit, the glitch may be passed along by the transistors and interconnect (transport) in the circuit, or it may be swallowed and not passed (internal) to the next resource in the FPGA. This depends on the width of the glitch and the type of resource the glitch passes through. To produce more accurate simulation of how signals are propagated within the silicon, Xilinx® models this behavior in the timing simulation netlist. For Verilog simulation, this information is passed by the PATHPULSE construct in the SDF file. This construct is used to specify the size of pulses to be rejected or swallowed on components in the netlist. For VHDL, there are two library components called X_BUF_PP and X_TRI_PP in which proper values are annotated for pulse rejection in the simulation netlist. The result of these constructs in our simulation netlists is a more true-to-life simulation model, and so a more accurate simulation.

## CLKDLL/DCM Clocks do not appear de-skewed

The CLKDLL and DCM components remove the clock delay from the clock entering into the chip. As a result, the incoming clock and the clocks feeding the registers in the device have a minimal skew within the range specified in the Databook for any given device. However, in timing simulation, the clocks may not appear to be de-skewed within the range specified. This is due to the way the delays in the SDF file are handled by some simulators.

The SDF file annotates the CLOCK PORT delay on the X_FF components. However, some simulators may show the clock signal in the waveform viewer before taking this delay into account. If it appears that the simulator is not properly de-skewing the clock, consult your simulator's documentation to find out if it is not displaying the input port delays in the waveform viewer at the input nodes. If this is the case, then when the CLOCK PORT delay on the X_FF is added to the internal clock signal, it should line up within the device specifications in the waveform viewer with the input port clock. The simulation is still functioning properly, the waveform viewer is just not displaying the signal at the expected node. To verify that the CLKDLL / DCM is functioning correctly, delays from the SDF file may need to be accounted for manually to calculate the actual skew between the input and internal clocks.

## Simulating the DLL/DCM

Although the functionality of the Xilinx® DLL and DCM components may seem easy to understand, the simulation of these components can be easily misinterpreted. The purpose of this section is to clarify how the DLL / DCM is supposed to simulate, and to identify some of the common problems designers face when simulating these components.

### TRACE/Simulation Model Differences

To fully understand the simulation model, you must first understand that there are differences in the way the DLL / DCM is built in silicon, the way TRACE reports their timing and the way the DLL / DCM is modeled for simulation. The DLL / DCM simulation model attempts to replicate the functionality of the DLL / DCM in the Xilinx® silicon, but it does not always do it exactly how it is implemented in the silicon. In the silicon, the DLL / DCM uses a tapped delay line to delay the clock signal. This accounts for input delay paths and global buffer delay paths to the feedback in order to accomplish the proper clock phase adjustment. TRACE or Timing Analyzer reports the phase adjustment as a simple delay (usually negative) so that you can adjust the clock timing for static timing analysis. As for simulation, the DLL / DCM simulation model itself attempts to align the input clock to the clock coming back into the feedback input. Instead of putting the delay in the DLL or DCM itself, the delays are handled by combining some of them into the feedback path as clock delay on the clock buffer (component) and clock net (port delay). The remainder is combined with the port delay of the CLKFB pin. While this is different from the way TRACE or Timing Analyzer reports it, and the way it is implemented in the silicon, the end result is the same functionality and timing. TRACE and simulation both use a simple delay model rather than an adjustable delay tap line similar to silicon.

The primary job of the DLL / DCM is to remove the clock delay from the internal clocking circuit as shown in the following figure.

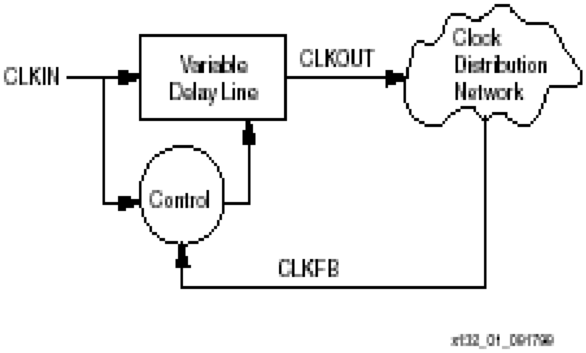


*Figure 6-7:* **Delay Locked Loop Block Diagram**

Do not confuse this with the function of de-skewing the clock. Clock skew is generally associated with delay variances in the clock tree, which is a different matter. By removing the clock delay, the input clock to the device pin should be properly phase aligned with the clock signal as it arrives at each register it is sourcing. This means that observing signals at the DLL / DCM pins generally does not give the proper view point to observe the removal of the clock delay. The place to see if the DCM is doing its job is to compare the input clock (at the input port to the design) with the clock pins of one of the sourcing registers. If these are aligned (or shifted to the desired amount) then the DLL / DCM has accomplished its job.

## Non-LVTTL Input Drivers

When using non-LVTTL input buffer drivers to drive the clock, the DCM does not make adjustments as to the type of input buffer chosen, but instead has a single delay value to provide the best amount of clock delay across all I / O standards. If you are using the same input standard for the data, the delay values should track, and generally not cause a problem. Even if you are not using the same input standard, the amount of delay variance generally does not cause hold time failures because the delay variance is small compared to the amount of input delay. The delay variance is calculated in both static timing analysis and simulation so you should see proper setup time values during static timing analysis, as well as during simulation.

## Viewer Considerations

Depending on which simulator you use, the waveform viewer might not depict the delay timing the way you expect to see it. Some simulators, including the current version of MTI® ModelSim™, combine interconnect delays (either interconnect or port delays) with the input pins of the component delays when you view the waveform on the waveform viewer. In terms of the simulation, the results are correct, but in terms of what you see in the waveform viewer, this may not always be what you expect to see.

Since interconnect delays are combined, when you look at a pin using the MTI® ModelSim™ viewer, you do not see the transition as it happens on the pin. In terms of functionality, the simulation acts properly, and this is not very relevant, but when attempting to calculate clock delay, the interconnect delays before the clock pin must be taken into account if the simulator you are using combines these interconnect delays with component delays. Please refer to Answer Record 11067 on the Xilinx® support website at http://support.xilinx.com for the most current information on this issue.

### Attributes for Simulation and Implementation

Ensure that the same attributes are passed for simulation and implementation. During implementation of the design, DLL / DCM attributes may be passed either by the synthesis tool (via a synthesis attribute, generic or defparam declaration), or within the UCF file. For RTL simulation of the UNISIM models, the simulation attributes must be passed via a generic if you are using VHDL, or a defparam if you are using Verilog. If you do not use the default setting for the DLL / DCM, and you use a UCF file or a synthesis attribute to pass the attribute values, you must ensure that the attributes for RTL simulation are the same as those used for implementation. If not, there may be differences between RTL simulation and the actual device implementation. The best way to ensure the attributes passed to implementation are the same as those used for simulation is to use the generic mapping method (VHDL) or defparam passing (Verilog) if your synthesis tool supports these methods for passing functional attributes.

## Simulating the DCM in Digital Frequency Synthesis Mode Only

To simulate the DCM in Digital Frequency Synthesis Mode only, set the CLK_FEEDBACK attribute to NONE and leave the CLKFB unconnected. The CLKFX and CLKFX180 are generated based on CLKFX_MULTIPLY and CLKFX_DIVIDE attributes. These outputs do not have phase correction with respect to CLKIN.

# Debugging Timing Problems

In back-annotated (timing) simulation, the simulator takes into account timing information that resides in the standard delay format (SDF) file. This may lead to eventual timing violations issued by the simulator if the circuit is operated too fast or has asynchronous components in the design. This section explains some of the more common timing violations, and gives advice on how to debug and correct them.

## Identifying Timing Violations

After you run timing simulation, check the messages generated by your simulator. If you have timing violations, they are indicated by warning or error messages.

The following example is a typical setup violation message from MTI® ModelSim™ for a Verilog design. Message formats vary from simulator to simulator, but all contain the same basic information. See your simulator documentation for details.

```
# ** Error:/path/to/xilinx/verilog/src/simprims/X_RAMD16.v(96):
$setup(negedge WE:29138 ps, posedge CLK:29151 ps, 373 ps);
# Time:29151 ps Iteration:0 Instance: /test_bench/u1/\U1/X_RAMD16\
```

- The first line points to the line in the simulation model that is in error. In the example above, the failing line would be line 96 of the Verilog file, X_RAMD16.

- The second line gives information about the two signals that are the cause of the error. This line states the following.
    - The type of violation ($setup, $hold, $recovery, etc.). The above example is a $setup violation.
    - The name of each signal involved in the violation followed by the simulation time at which that signal last changed values. In the above example, the failing signals would be the negative-going edge of the signal WE, which last changed at 29138 picoseconds, and the positive-going edge of the signal CLK, which last changed at 29151 picoseconds.
    - The third value is the allotted amount of time for the setup. For this example, the signal on WE should be ′stable for 373 pico seconds before the clock transitions. Since WE changed only 13 pico seconds before the clock, this violation was reported.
- The third line gives the simulation time at which the error was reported, and the instance in the structural design (time_sim) in which the violation occurred.

## Verilog System Timing Tasks

Verilog system tasks and functions are used to perform simulation related operations such as monitoring and displaying simulation time and associated signal values at a specific time during simulation. All system tasks and functions begin with a dollar sign, for example $setup. See the Verilog Language Reference Manual (available from IEEE) for details about specific system tasks.

Timing check tasks may be invoked in specific blocks to verify the timing performance of a design by making sure critical events occur within given time limits. Timing checks perform the following steps:

- Determine the elapsed time between two events.
- Compare the elapsed time to a specified limit.
- If the elapsed time does not fall within the specified time window, report timing violation.

The following system tasks may be used for performing timing checks:

$hold
$nochange
$period
$recovery
$setup
$setuphold
$skew
$width

## VITAL Timing Checks

VITAL (VHDL Initiative Towards ASIC Libraries) is an addition to the VHDL specification that deals with adding timing information to VHDL models. One of the key aspects of VITAL is the specification of the package vital_timing. This package, in addition to other things, provides standard procedures for performing timing checks.

The package vital_timing defines the following timing check procedures:

- VitalSetupHoldCheck
- VitalRecoveryRemovalCheck
- VitalInPhaseSkewCheck

- VitalOutPhaseSkewCheck
- VitalPeriodPulseCheck.

VitalSetupHoldCheck is overloaded for use with test signals of type Std_Ulogic or Std_Logic_Vector. Each defines a CheckEnabled parameter that supports the modeling of conditional timing checks. See the *VITAL Language Reference Manual* (available from IEEE) for details about specific VITAL timing checks.

## Timing Problem Root Causes

Timing violations, such as $setuphold, occur any time data changes at a register input (either data or clock enable) within the setup or hold time window for that particular register. There are a few typical causes for timing violations; the most common are the following.

- The design is not constrained
- A path in the design is not constrained
- The design does not meet timespecs
- The design simulation clock does not match what is called for in the timespecs
- Clock skew is unaccounted for in a particular data path
- A path in the design has asynchronous inputs, crosses out-of-phase clock domains or has asynchronous clock boundaries

### Design Not Constrained

Timing constraints are essential to help you meet your design goals or obtain the best implementation of your circuit. Global timing constraints cover most constrainable paths in a design. These global constraints cover clock definitions, input and output timing requirements, and combinatorial path requirements. Specify global constraints like PERIOD, OFFSET_IN_BEFORE, and OFFSET_OUT_AFTER to match your simulation stimulus with the timespecs of the devices used in the design.

In general, keep in mind the following two points when constraining a design:

- PERIOD: Can be quickly applied to a design. It also leads in the support of OFFSET, which you can use to specify your I/O timing. This works well for a single-clock design , or multi-clock design that is not multi-cycle.
- FROM-TO: This constraint works well with more complicated timing paths. Designs that are multi-cycle or have paths that cross clock domains are better handled this way. For I/O, however, you must add/subtract the delay of the global buffer. Note that using an OFFSET before for input and an OFFSET after for output is supported without the need to specify a period, so you can use the advantages of both.

For detailed information on constraining your design, consult any or all of the following resources.

- *Constraints Guide*:

  The *Constraints Guide* lists all of the Xilinx® constraints along with explanations and guides to their usage.

  The Timing Constraint Strategies chapter in the *Constraints Guide* gives detailed information on the best ways to constrain the timing on your design to get optimum results.

- *Timing and Constraints* area on the Xilinx® home page:

  The *Timing and Constraints* area on the Xilinx® home page provides a presentation of *Basic Timing Concepts and Syntax Examples.* This presentation gives an overview of how to constrain your design, and has examples of how to code various constraints.

- The Timing Improvement Wizard*:*

  The Timing Improvement Wizard provides suggestions for improving failing paths, and can help you find answers to your specific timing questions. You can find the Timing Improvement Wizard at:

  http://support.xilinx.com/support/troubleshoot/psolvers.htm

## Path Not or Improperly Constrained

Unconstrained or improperly constrained data and clock paths are the most common sources of setup and hold violations. Because data and clock paths can cross domain boundaries, global constraints are not always adequate to ensure that all paths are constrained. For example, a global constraint, such as PERIOD, does not constrain paths that originate at an input pin, and data delays along these paths could cause setup violations.

Use Timing Analyzer to determine the length of an individual data or clock path. For input paths to the design, if the length of a data path minus the length of the corresponding clock path, plus any data delay, is greater than the clock period, you get a setup violation.

```
clock period < data path - clock path + data delay value setup value for
register
```

For detailed information on constraining paths driven by input pins, see the Timing Constraint Strategies chapter of the *Constraints Guide.* Also see "Design Not Constrained" above for other constraints resources.

## Design Does Not Meet Timespec

Xilinx® software enables you to specify precise timing requirements for your Xilinx® FPGA designs. Specify the timing requirements for any nets or paths in your design. The primary method of specifying timing requirements is by assigning timing constraints. You can enter timing constraints through your synthesis tool, the Xilinx® Constraints Editor, or by editing the User Constraint File (UCF). For detailed information on entering timing specifications, see the *Development System Reference Guide*. For detailed information about the constraints you can use with your schematic entry software, see the *Constraints Guide*.

Once you define timing specifications, use TRACE (Timing Report, Circuit Evaluator, and TSI Report) or Timing Analyzer to analyze the results of your timing specifications. Review the timing report carefully to ensure that all paths are constrained, and that the constraints are specified properly. Be sure to check for any error messages in the report.

If after applying timing constraints your design still does not meet timespec, there are several things you can do. Generally, your synthesis and implementation tools have options intended to improve timing performance. Check with your tool's documentation to see what options can be applied to your design.

If refining your tool options is not sufficient, it may be necessary to go back to your source code to reconfigure parts of your design. Reducing levels of logic reduces timing delays, as well as arranging your floor plan so that related logic is grouped together.

## Simulation Clock Does Not Meet Timespec

If the frequency of the clock that was specified during simulation is greater than that specified in the timing constraints, then this over-clocking of the design could cause timing violations. For example, if your simulation clock has a frequency of 5 ns, and you have a PERIOD constraint set at 10 ns, a timing violation could occur. This situation can also be complicated by the presence of DLL or DCM in the clock path.

Generally, this problem is caused by an error either in the test bench or in the constraint specification. Check to ensure that the constraints match the conditions in the test bench, and correct any inconsistencies. If you modify the constraints, be sure to re-run the design through place and route to ensure that all constraints are met.

## Unaccounted Clock Skew

Clock skew is the difference between the amount of time the clock signal takes to reach the destination register, and the amount of time the clock signal takes to reach the source register. The data must reach the destination register within a single clock period plus or minus the amount of clock skew. Clock skew is generally not a problem when you use global buffers; however, clock skew can be a concern if you use the local routing network for your clock signals.

To find out if clock skew is your problem, use TRACE to do a setup test. See the TRACE chapter of the *Development Systems Reference Guide* for directions on how to run a setup check, and read the TRACE report. You can also use Timing Analyzer to determine clock skew. See the *Timing Analyzer Online Help* for instructions.

Be aware that clock skew is modeled in the simulation, but not in TRACE unless you invoke TRACE using the "-skew" switch. Simulation results may not equal TRACE results if the skew is significant (as when a non-BUFG clock is used). To account for skew in TRACE, use the following command:

```
trce -skew
```

or set the following environment variable:

```
setenv XILINX_DOSKEWCHECK yes
```

If your design has clock skew, consider redesigning your path so that all registers are tied to the same global buffer. If that is not possible, consider using the USELOWSKEWLINES constraint to minimize skew. Refer to the *Constraints Guide* for detailed information on the USELOWSKEWLINES constraint.

***Note:*** Avoid using the `XILINX_DOSKEWCHECK` environment variable with PAR. If you have clocks on local routing, the PAR timing score may oscillate. This is because the timing score is a function of both a clock delay and the data delay, and attempts to make the data path faster may make the clock path slower, or vice versa. It should only be used within PAR on designs with paths that make use of global clock resources.

### Asynchronous Inputs, Asynchronous Clock Domains, Crossing Out-of-Phase

Timing violations can be caused by data paths that are not controlled by the simulation clock, or are not clock controlled at all. Timing violations also include data paths that cross asynchronous clock boundaries, have asynchronous inputs, or cross data paths out of phase.

- Asynchronous Clocks

  If the design has two or more clock domains defined, any path that crosses data from one domain to another could cause timing problems. Although data paths that cross from one clock domain to another are not always asynchronous, it is always best to be cautious with these situations. If two clocks have unrelated frequencies, they should certainly be treated as asynchronous. Any clocking signal that is coming from off-chip should also be treated as asynchronous. Note that any time a register's clock is gated, it should be treated as asynchronous unless extreme caution is used.

  Check the source code and the Timing Analyzer report to see if the path in question crosses asynchronous clock boundaries. If your design does not allow enough time for the path to be properly clocked into the other domain, you may have to redesign your clocking scheme. Consider using an asynchronous FIFO as a better way to pass data from one clock domain to another.

- Asynchronous Inputs

  Data paths that are not controlled by a clocked element are asynchronous inputs. Because they are not clock controlled, they can easily violate setup and hold time specifications.

  Check the source code to see if the path in question is synchronous to the input register. If synchronization is not possible, you can use the ASYNC_REG constraint to work around the problem. See "Using the ASYNC_REG Attribute" in this chapter.

- Out of Phase Data Paths

  Data paths can be clock controlled at the same frequency, but nevertheless can have setup or hold violations because the clocks are out of phase. Even if the clock frequencies are a derivative of each other, improper phase alignment could cause setup violations.

  Check the source code and the Timing Analyzer report to see if the path in question crosses another path with an out of phase clock.

## Debugging Tips

When you are faced with a timing violation, the following questions may give valuable clues as to what went wrong.

- Was the clock path analyzed by TRACE or Timing Analyzer?
- Did TRACE or Timing Analyzer report that the data path can run at speeds being clocked in simulation?
- Is clock skew being accounted for in this path delay?
- Does subtracting the clock path delay from the data path delay still allow clocking speeds?
- Will slowing down the clock speeds eliminate the $setup / $hold time violations?

- Does this data path cross clock boundaries (from one clock domain to another)? Are the clocks synchronous to each other? Is there appreciable clock skew or phase difference between these clocks?
- If this path is an input path to the device, does changing the time at which the input stimulus is applied eliminate the $setup / $hold time violations?

Based on the answers to these questions, you may need to make changes to your design or test bench to accommodate the simulation conditions.

# Special Considerations for Setup and Hold Violations

## Zero Hold Time Considerations

While Xilinx® data sheets report that there are zero hold times on the internal registers and I/O registers with the default delay and using a global clock buffer, it is still possible to receive a $hold violation from the simulator. This $hold violation is really a $setup violation on the register. However, in order to get an accurate representation of the CLB delays, part of the setup time must be modeled as a hold time. For more information on this modeling, please refer to Xilinx® Answer Record 782 at the Xilinx® support website.

## Negative Hold Times

In older versions of Xilinx® simulation models, negative hold times were truncated and specified as zero hold times. While this does not cause inaccuracies for simulation, it does reveal a more pessimistic model in terms of timing than is possible in the actual FPGA. Therefore, this made it more difficult to meet stringent timing requirements. With the current release, negative hold times are now specified in the timing models to provide a wider, yet more accurate representation of the timing window. This is accomplished by combining the setup and hold parameters for the synchronous models into a single setuphold parameter in which the timing for the setup / hold window can be expressed. This should not change the timing simulation methodology in any way; however, when using Cadence™ NC-Verilog™, there are no longer separate violation messages for setup and hold as they are now combined into a single setuphold violation.

## RAM Considerations

### Timing Violations

Xilinx® devices contain two types of memories, Block RAM and Distributed RAM. Both Block RAM and Distributed RAM are synchronous elements when you write data to them, so the same precautions must be taken as with all synchronous elements to avoid timing violations. The data input, address lines, and enables all must be stable before the clock signal arrives to guarantee proper data storage.

### Collision Checking

Block RAMs also perform synchronous read operations. This means that during a read cycle, the addresses and enables must be stable before the clock signal arrives or a timing violation may occur.

When using Distributed RAM or Block RAM in dual-port mode, special care must be taken to avoid memory collisions. A memory collision occurs when one port is being written to while the other port is either read or write is attempted to the same address at the same time, or within a very short period of time thereafter. The model warns you if a collision occurs. If the RAM is being read on one port as it is being written to on the other, the model outputs an 'X' value signifying an unknown output. If the two ports are writing data to the same address at the same time, the model can write unknown data into memory. Special care should be taken to avoid this situation as unknown results may occur from this action. For the hardware documentation on collision checking, refer to the Design Considerations chapter, Using Block SelectRAM™ Memory section of the *Virtex-II Platform FPGA User Guide*.

### Hierarchy Considerations

It is possible for the top-level signals to switch correctly, keeping the setup and hold times accounted for, and at the same time, have an error reported at the lowest level primitive in the design. This can happen because as the signals travel down through the hierarchy to this low-level primitive, the delays they go through can reduce the differences between them to the point where they begin to violate the setup time.

To correct this problem, follow these steps.

1. Browse the design hierarchy, and add the signals of the instance reporting the error to the top-level waveform. Ensure that the setup time is indeed being violated at the lower level.
2. Step back through the structural design until a link between an RTL (pre-synthesis) design path and this instance reporting the error can be determined.
3. Constrain the RTL path using timing constraints so that the timing violation no longer occurs. Usually, most implemented designs have a small percentage of unconstrained paths after timing constraints have been applied, and these are the ones where $setup and $hold violations generally occur.

***Note:*** The debugging steps for the $hold violations and the $setup violations are identical.

## $Width Violations

The $width Verilog system task monitors the width of signal pulses. When the pulse width of a specific signal is less than what is required for the device being used, the simulator issues a $width violation. Generally, $width violations are only specified for clock signals and asynchronous set or reset signals.

Consult the specific product Data Sheets for the device switching characteristics for your device. Find the minimum pulse width requirements, and ensure that the device stimulus conforms to these specifications.

## $Recovery Violations

The $recovery Verilog system task specifies a time constraint between an asynchronous control signal and a clock signal (for example, between clearbar and the clock for a flip-flop). A $recovery violation occurs when a change to the signal occurs within the specified time constraint.

The $recovery Verilog system task is used to check for one of two dual-port block RAM conflicts:

- If both ports write to the same memory cell simultaneously, violating the clock-to-setup requirement, the data stored is invalid.
- If one port attempts to read from the same memory cell to which the other is simultaneously writing (also violating the clock setup requirement) the write will be successful, but the data read will be invalid.

Recovery tasks are also used to detect if an asynchronous set / reset signal is released just before a clock event occurs. If this happens, the result is similar to a setup violation in that it is undetermined whether the new data should be clocked in or not.

# Simulation Flows

When simulating, compile the Verilog source files in any order since Verilog is compile order independent. However, VHDL components must be compiled bottom-up due to order dependency. Xilinx® recommends that you specify the test fixture file before the HDL netlist of your design, as in the following examples.

Xilinx® recommends giving the name *testbench* to the main module in the test fixture file. This name is consistent with the name used by default in the ISE Project Navigator. If this name is used, no changes are necessary to the option in ISE in order to perform simulation from that environment.

## ModelSim™ Vcom

The following is information regarding ModelSim™ Vcom.

### Using Shared Precompiled Libraries

Simulation Libraries have to be compiled to *compiled_lib_dir* before using ModelSim™ Vcom. See "Compiling Xilinx® Simulation Libraries (COMPXLIB)" for instruction on how to compile the Xilinx® VHDL libraries.

Depending on the makeup of the design (Xilinx® instantiated primitives, or CORE Generator™ components), for RTL simulation, specify the following at the command line.

1. Create working directory.

   ```
   vlib work
   ```

2. Compile design files and workbench.

   ```
   vcom lower_level_files.vhd top_level.vhd testbench.vhd
       (testbench_cfg.vhd)
   ```

3. Simulate design.

   ```
   vsim testbench_cfg
   ```

For timing or post-NGDBuild simulation, use the SIMPRIM-based libraries. Specify the following at the command line:

1. Create working directory.

   ```
   vlib work
   ```

2. Compile design files and workbench.

   ```
   vcom design.vhd testbench.vhd [testbench_cfg.vhd]
   ```

3. Simulate design.

   ```
   vsim -sdfmax instance_name=design.sdf testbench_cfg
   ```

## Scirocco™

### Using Shared Precompiled Libraries

Simulation Libraries have to be compiled to *compiled_lib_dir* before using Scirocco™. See "Compiling Xilinx® Simulation Libraries (COMPXLIB)" for instruction on how to compile the Xilinx® VHDL libraries.

Depending on the makeup of the design (Xilinx® instantiated components, or CORE Generator™ components), for RTL simulation, specify the following at the command line.

1. Create working directory.

```
mkdir work
```

2. Compile design files and workbench.

```
vhdlan work_macro1.vhd top_level.vhd testbench.vhd
   testbench_cfg.vhd
scs testbench_cfg
```

3. Simulate design.

```
scsim
```

For timing or post-NGDBuild simulation, use the SIMPRIM-based libraries. Specify the following at the command line:

1. Create working directory.

```
mkdir work
```

2. Compile design files and workbench.

```
vhdlan work_design.vhd testbench.vhd
scs testbench
```

3. Simulate design.

```
scsim -sdf testbench:design.sdf
```

## NC-VHDL™

The following is information regarding NC-VHDL™.

### Using Shared Precompiled Libraries

Simulation Libraries have to be compiled to *compiled_lib_dir* before using NC_VHDL. See "Compiling Xilinx® Simulation Libraries (COMPXLIB)" for instruction on how to compile the Xilinx® VHDL libraries. It is assumed that the proper mapping and setup files are present before simulation. If you are unsure that you have the simulator properly set up, please consult the simulator vendor documentation. Depending on the makeup of the design (Xilinx® instantiated components, or CORE Generator™ components), for RTL simulation, specify the following at the command line.

1. Create a working directory.

```
mkdir test
```

2. Compile design files and workbench.

```
ncvhdl -work test testwork_macro1.vhd top_level.vhd testbench.vhd
     testbench_cfg.vhd
```

3. Elaborate the design at the proper scope.

```
ncelab testbench_cfg:A
```

4. Invoke the simulation.

   ```
   ncsim testbench_cfg:A
   ```

For timing or post-NGDBuild simulation, use the SIMPRIM-based libraries. Specify the following at the command line:

1. Compile the SDF annotation file:

   ```
   ncsdfc design.sdf
   ```

2. Create an SDF command file, `sdf.cmd`, the following data in it:

   ```
   COMPILED_SDF_FILE = design.sdf.X
   SCOPE = uut,
   MTM_CONTROL = 'MAXIMUM';
   ```

3. Create a working directory.

   ```
   mkdir test
   ```

4. Compile design files and workbench.

   ```
   ncvhdl -work test work_design.vhd testbench.vhd
   ```

5. Elaborate the design at the proper scope.

   ```
   ncelab -sdf_cmd_file.cmd testbench_cfg:A
   ```

6. Invoke the simulation.

   ```
   ncsim testbench_cfg:A
   ```

## NC-Verilog™

There are two methods to run simulation with NC-Verilog™.

1. Using library source files with compile time options.
2. Using shared precompiled libraries.

### Using Library Source Files With Compile Time Options

Depending on the makeup of the design (Xilinx® instantiated primitives, or CORE Generator™ components), for RTL simulation, specify the following at the command line:

```
ncxlmode +libext+.v -y $XILINX/verilog/src/unisims \
    testfixture.v design.v $XILINX/verilog/src/glbl.v
```

For timing or post-NGDBuild simulation, use the SIMPRIM-based libraries. Specify the following at the command line.

```
ncxlmode -y $XILINX/verilog/src/simprims \
    +libext+.v testfixture.v time_sim.v $XILINX/verilog/src/glbl.v
```

### Using Shared Precompiled Libraries

Simulation Libraries have to be compiled to *compiled_lib_dir* before using NC-Verilog™. See "Compiling Xilinx® Simulation Libraries (COMPXLIB)" for instruction on how to compile the Xilinx® Verilog libraries.

Depending on the makeup of the design (Xilinx® instantiated primitives, or CORE Generator™ components), for RTL simulation, edit the `hdl.var` and `cds.lib` files to specify the library mapping.

```
# cds.lib
DEFINE simprims_ver compiled_lib_dir/simprims_ver
DEFINE xilinxcorelib_ver compiled_lib_dir/xilinxcorelib_ver
DEFINE worklib worklib

# hdl.var
DEFINE VIEW_MAP ($VIEW_MAP, .v => v)
DEFINE LIB_MAP ($LIB_MAP, compiled_lib_dir/unisims_ver => unisims_ver)
DEFINE LIB_MAP ($LIB_MAP, compiled_lib_dir/simprims_ver => simprims_ver)
DEFINE LIB_MAP ($LIB_MAP, + => worklib)
// After setting up the libraries, now compile and simulate the design:

ncvlog -messages -update $XILINX/verilog/src/glbl.v testfixture.v design.v
ncelab -messages testfixture_name glbl
ncsim -messages testfixture_name
```

The –update option of Ncvlog enables incremental compilation.

For timing or post-NGDBuild simulation, use the SIMPRIM-based libraries. Specify the following at the command line:

```
ncvlog -messages -update $XILINX/verilog/src/glbl.v testfixture.v time_sim.v
ncelab -messages -autosdf testfixture_name glbl
ncsim -messages testfixture_name
```

For more information on how to back-annotate the SDF file for timing simulation, go to Answer Record 941 on the Xilinx® support website at http://support.xilinx.com.

## VCS™/VCSi™

VCS™ and VCSi™ are identical except that VCS™ is more highly optimized, resulting in greater speed for RTL and mixed level designs. Pure gate level designs run with comparable speed. However, VCS™ and VCSi™ are guaranteed to provide the exact same simulation results. VCSi™ is invoked using the **vcsi** command rather than the **vcs**. command.

There are two methods to run simulation with VCS™/VCSi™.

1. Using library source files with compile time options.
2. Using shared precompiled libraries.

### Using Library Source Files With Compile Time Options

Depending on the makeup of the design (Xilinx® instantiated primitives, or CORE Generator™ components), for RTL simulation, specify the following at the command line.

```
vcs -y $XILINX/verilog/src/unisims \
    +libext+.v $XILINX/verilog/src/glbl.v \
    -Mupdate -R testfixture.v design.v
```

For timing or post-NGDBuild, use the SIMPRIM-based libraries. Specify the following at the command line.

```
vcs +compsdf -y $XILINX/verilog/src/simprims \
    $XILINX/verilog/src/glbl.v \
    +libext+.v -Mupdate -R testfixture.v time_sim.v
```

The –R option automatically simulates the executable after compilation.

The –Mupdate option enables incremental compilation. Modules are recompiled for any of the following reasons:

1. Target of a hierarchical reference has changed.
2. A compile time constant, such as a parameter, has changed.
3. Ports of a module instantiated in the module have changed.
4. Module inlining. For example, a group of module definitions merging, internally in VCS™ into a larger module definition which leads to faster simulation. These affected modules are again recompiled. This is done only once.

For more information on how to back-annotate the SDF file for timing simulation, go to Answer Record 6349 on the Xilinx® support website at http://support.xilinx.com.

## Using Shared Precompiled Libraries

Simulation Libraries have to be compiled to *compiled_lib_dir* before using VCS™/VCSi™. See "Compiling Xilinx® Simulation Libraries (COMPXLIB)" for instruction on how to compile the Xilinx® Verilog libraries.

Depending on the makeup of the design (Xilinx® instantiated primitives, or CORE Generator™ components), for RTL simulation, specify the following at the command line.

```
vcs -Mupdate -Mlib=compiled_dir/unisims_ver -y \
    $XILINX/verilog/src/unisims -Mlib=compiled_dir/simprims_ver -y \
    $XILINX/verilog/src/simprims \
    -Mlib=compiled_dir/xilinxcorelib_ver +libext+.v \
    $XILINX/verilog/src/glbl.v -R testfixture.v design.v
```

For timing or post-NGDBuild simulation, the SIMPRIM-based libraries are used. Specify the following at the command line.

```
vcs +compsdf -y $XILINX/verilog/src/simprims \
    $XILINX/verilog/src/glbl.v +libext+.v-Mupdate -R \
    testfixture.v time_sim.v
```

The –R option automatically simulates the executable after compilation. Finally, the –Mlib=*compiled_lib_dir* option provides VCS™ with a central place to look for the descriptor information before it compiles a module and a central place to get the object files when it links together the executable.

The –Mupdate option enables incremental compilation. Modules are recompiled for any of the following reasons:

1. Target of a hierarchical reference has changed.
2. A compile time constant such as a parameter has changed.
3. Ports of a module instantiated in the module have changed.
4. Module inlining. For example, a group of module definitions merging, internally in VCS™, into a larger module definition which leads to faster simulation. These affected modules are again recompiled. This is done only once.

For more information on how to back-annotate the SDF file for timing simulation, go to Answer Record 6349 on the Xilinx® support website at http://support.xilinx.com.

## ModelSim™ Vlog

There are two methods to run simulation with ModelSim™ Vlog.

1. Using library source files with compile time options.
2. Using shared precompiled libraries.

### Using Library Source Files With Compile Time Options

Depending on the makeup of the design (Xilinx® instantiated primitives, or CORE Generator™ components), for RTL simulation, specify the following at the ModelSim™ prompt:

```
set XILINX $env(XILINX)
vlog -y $XILINX/verilog/src/unisims \
    +libext+.v $XILINX/verilog/src/glbl.v -incr testfixture.v design.v
    vsim testfixture glbl
```

For timing or post-NGDBuild simulation, the SIMPRIM-based libraries are used. Specify the following at the ModelSim™ prompt:

```
vlog -y $XILINX/verilog/src/simprims $XILINX/verilog/src/glbl.v \
    +libext+.v testfixture.v time_sim.v -incr
    vsim testfixture glbl +libext+.v testfixture.v
```

The **-incr** option enables incremental compilation.

### Using Shared Precompiled Libraries

Simulation Libraries have to be compiled to *compiled_lib_dir* before using ModelSim™ Vlog. See "Compiling Xilinx® Simulation Libraries (COMPXLIB)" for instruction on how to compile the Xilinx® Verilog libraries.

Depending on the makeup of the design (Xilinx® instantiated primitives, or CORE Generator™ components), for RTL simulation, specify the following at the ModelSim™ prompt:

```
set XILINX $env(XILINX)
vlog $XILINX/verilog/src/glbl.v testfixture.v time_sim.v -incr
vsim -L unisims_ver -L simprims_ver -L xilinxcorelib_ver \
    testfixture glbl
```

For timing or post-NGDBuild simulation, the SIMPRIM-based libraries are used. Specify the following at the ModelSim™ prompt:

```
vlog $XILINX/verilog/src/glbl.v testfixture.v time_sim.v -incr
vsim -L simprims_ver testfixture glbl
```

The –incr option enables incremental compilation. The –L *compiled_lib_dir* option provides VSIM with a library to search for design units instantiated from Verilog.

# IBIS

The Xilinx® IBIS models provide information on I/O characteristics. The IBIS models can be used for the following.

IBIS models provide information about I/O driver and receiver characteristics without disclosing proprietary knowledge of the IC design (as unencrypted SPICE models do). However, there are some limitations on the information that IBIS models can provide. Please note that these are limitations imposed by the IBIS specification itself.

IBIS models can be used for the following:

1. Model best-case and worst-case conditions (best-case = strong transistors, low temperature, high voltage; worst-case = weak transistors, high temperature, low voltage). Best-case conditions are represented by the "fast/strong" model, while worst-case conditions are represented by the "slow/weak" model. Typical behavior is represented by the "typical" model.
2. Model varying drive strength and slew rate conditions for Xilinx® I/Os that support such variation.

IBIS ***cannot*** be used for any of the following:

1. Provide internal timing information (propagation delays and skew).
2. Model power and ground structures.
3. Model pin-to-pin coupling.
4. Provide detailed package parasitic information. Package parasitics are provided in the form of lumped RLC data. This is typically not a significant limitation, as package parasitics have an almost negligible effect on signal transitions.

The implications of (2) and (3) above are that ground bounce, power supply droop, and simultaneous switching output (SSO) noise CANNOT be simulated with IBIS models. To ensure that these effects do not harm the functionality of your design, Xilinx® provides device/package-dependent SSO guidelines based on extensive lab measurements. The locations of these guidelines are as follows:

- Virtex-II™ — The "Design Considerations" section of the *Virtex-II Platform FPGA User Guide*: http://support.xilinx.com/publications/products/ug_index.htm
- Virtex-II Pro™ — The "PCB Design Considerations" section of the *Virtex-II Pro Platform FPGA User Guide*: http://support.xilinx.com/publications/products/ug_index.htm
- Virtex™/-E — Xilinx® Application Note 133: "Using the Virtex™ Select I/O Resource" (XAPP133 Application Note)
- Spartan™-II/IIE — Xilinx® Application Note 179: "Using Select I/O Interfaces in Spartan-II™ FPGAs" (XAPP179 Application Note)
- IBIS models for Xilinx® devices can be found at: http://support.xilinx.com/support/sw_ibis.htm
- For more information about the IBIS specification, please see the IBIS Home Page at http://www.eigroup.org/ibis/ibis.htm
- The Xilinx® IBIS models are available for download at: ftp://ftp.xilinx.com/pub/swhelp/ibis/

# STAMP

The Xilinx® development system supports Stamp Model Generation. This feature supports the use of board level Static Timing Analysis tools, such as Mentor Graphics' Tau and Mentor Graphic's Blast. With these tools, users of Xilinx® programmable logic products can accelerate board level design verification.

Using the –stamp switch in the Xilinx® program, Trace writes out the stamp models.

For more information on creating the STAMP files, options to use in Trace, and integrating it with Tau and Blast, please see the XAPP166 Application Note.

# Chapter 7

# Equivalency Checking

This chapter describes the basic Equivalency Checking flow using Xilinx® software. It includes the following sections.

- "Introduction"
- "Software and Device Support"
- "Equivalency Checking Compatible Simulation Models"
- "Setup for Synopsys® Formality™ and Verplex® Conformal™ FPGA"
- "Conformal™ FPGA and Synplify Pro™ Flow"
- "Formality™ and FPGA Compiler II™ Flow"
- "Conformal™ FPGA and FPGA Compiler II™ Flow"

## Introduction

With rapid increases in FPGA design sizes, new simulation and logic verification methodologies must be explored to expedite the verification of design logic and functionality. Because of its accuracy and speed, logic equivalency checking, also known as formal verification, is quickly gaining acceptance by designers creating multi-million gate designs. Formal verification requires the presence of a golden (verified) reference design, and checks the netlists from the design under development for logic equivalence with this golden standard. Typically, the golden reference design will be the RTL design, and it will be verified against the post-synthesis and post-place and route netlists. This will ensure that any transformations done by the synthesis tool or the Xilinx® implementation tools have not affected the functionality of the design.

## Software and Device Support

### Supported Equivalency Checking Software

Xilinx® supports the following equivalency checking software.

- Synopsys® Formality™
- Verplex® Conformal™ FPGA

### Supported Synthesis Tools

Xilinx® supports equivalency checking with the following synthesis tools.

- Synopsys® FPGA Compiler II™ (for use with Formality™ and Conformal™ FPGA)
- Synplicity® Synplify Pro™ 7.2 or later (for use with Conformal™ FPGA)

### Supported Devices

Xilinx® supports equivalency checking for the following technologies.

- Spartan-II™, Spartan-IIE™, Spartan-3™
- Virtex™, Virtex-E™, Virtex-II™, Virtex-II Pro™, Virtex-II Pro X™

## Equivalency Checking Compatible Simulation Models

There are two Xilinx® verification libraries that are used along with the equivalency checking (EC) tools. These are:

- **UNISIM –** The UNISIM library contains the Xilinx® primitives in RTL format. This library is required if the design contains any Xilinx® primitives, for example, an instantiation of a DCM or Block RAM. These libraries are located at:

  ```
  $XILINX/verilog/formality/unisims
  $XILINX/verilog/verplex/unisims
  ```

- **SIMPRIM –** The SIMPRIM library contains the Xilinx® Primitives for back-annotated Verification (post-NGDBuild, post-Map and post-PAR). Since the back-annotated netlist is composed completely of these gate-level primitives, this library needs to be read before verifying a post-NGDBuild, post-Map or post-PAR design. These libraries are located at:

  ```
  $XILINX/verilog/formality/simprims
  $XILINX/verilog/verplex/simprims
  ```

See the "Setup for Synopsys® Formality™" and "Setup for Verplex® Conformal™ FPGA"sections for information on how to read these libraries into the equivalency checking tools.

## Setup for Synopsys® Formality™ and Verplex® Conformal™ FPGA

This section describes the information needed to set up Formality™ or Conformal™ FPGA for verifying Xilinx® designs.

### Setup for Synopsys® Formality™

Following is a table of the environment variables that you must set before using Synopsys® Formality™ to verify a Xilinx® design.

*Table 7-1:* **Environment Variables Needed to Set Up Synopsys® Formality™ and Xilinx®**

| Name of Variable | Location Pointed by the Variable |
|---|---|
| **Synopsys® Variables** | |
| LM_LICENSE_FILE | `port@license_server` or `license_install_directory/license_file` |
| PATH | `formality_install_dir/fm/bin $path` |
| **Xilinx® Variables** | |
| XILINX | `xilinx_install_directory` |
| PATH | `$XILINX/bin/sol $path` |

Synopsys® also uses a setup file for each project. Create this setup file in the project directory by copying the template from `$XILINX/verilog/formality/template.synopsys_fm.setup` and renaming it to `.synopsys_fm.setup`. For more information on the use and customization possibilities of the `.synopsys_fm.setup` file, contact Synopsys®.

### Setup for Verplex® Conformal™ FPGA

Following is a table of the environment variables that you must set before using Verplex® Conformal™ FPGA to verify a Xilinx® design.

*Table 7-2:* **Environment Variables Needed to Set Up Verplex® Conformal™ FPGA and Xilinx®**

| Name of Variable | Location Pointed by the Variable |
|---|---|
| **Synopsys® Variables** | |
| VERPLEX_HOME | `conformal_lec_install_directory` |
| LM_LICENSE_FILE | `port@license_server` or `license_install_directory/license_file` |
| PATH | `$VERPLEX_HOME/bin $path` |
| **Xilinx® Variables** | |
| XILINX | `xilinx_install_directory` |
| PATH | `$XILINX/bin/sol $path` |

## Conformal™ FPGA and Synplify Pro™ Flow

When using Synplify Pro™ in verification mode and targeting a Xilinx® FPGA, you can use Conformal™ FPGA to verify that the golden RTL design and the post-synthesis netlist are logically equivalent. When verification mode is turned on, Synplify Pro™ produces a netlist for verification and several files that you can use to run the verification in Conformal™ FPGA. These files from Synplify Pro™ help automate the RTL to the post-synthesis verification process.

After verifying that the post-synthesis netlist is logically equivalent to the golden RTL design, the post-synthesis netlist is considered the golden netlist and can be compared against the post-place and route netlist to verify that it is also logically equivalent. Figure 7-1 shows the Conformal™ FPGA / Synplify Pro™ Flow for equivalency checking.

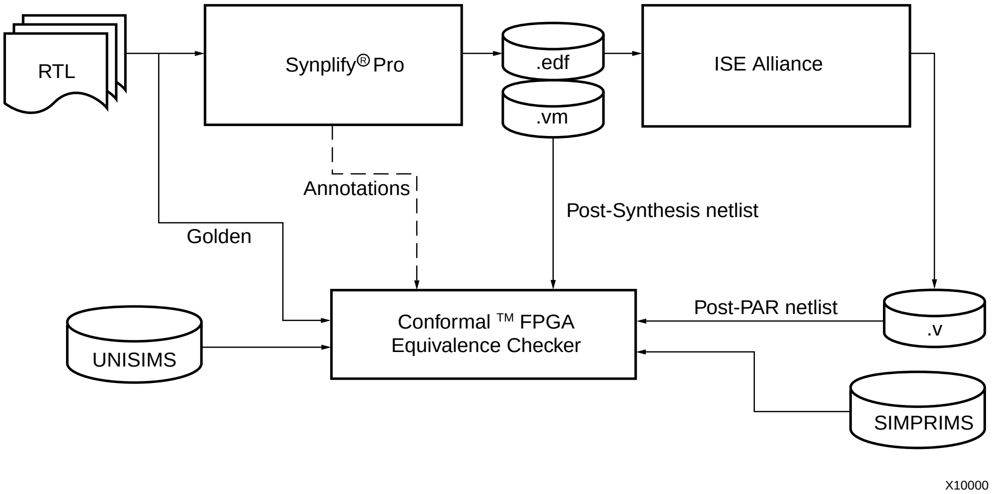


*Figure 7-1:* **Conformal™ FPGA and Synplify Pro™ Logic Equivalency Checking flow**

## RTL vs. Post-Synthesis

The first transformation of the RTL code occurs during synthesis. The synthesis tool optimizes the code and converts it from RTL to a post-synthesis netlist. Performing an equivalence check between the RTL and post-synthesis netlist verifies that the functionality of the design has not been changed by the transformation or by the optimization done by the synthesis tool.

***Note:*** In addition to the documentation provided in this guide, please see *Formal Verification Flow for Xilinx Devices Using the Synplify Pro Software and the Conformal_ LEC Tool* at the Symplicity website at http://www.synplicity.com/literature/pdf/formal_verification_final.pdf.

Perform the following steps to verify that the golden RTL and the post-synthesis netlist from Synplify Pro™ are logically equivalent.

1. Turn on verification mode in Synplify Pro™.
2. Run the DO file, `fpgaR2G.do`, created by Verplex® in Conformal™ FPGA.
3. If the two netlists are not equivalent, use Conformal™ FPGA to debug the differences. The debug process is not covered in this document. Please see the Conformal™ FPGA documentation for more information on using Conformal.

### Turning on Verification Mode

Do one of the following to turn on Verification Mode in Synplify Pro™:

- Command line:

```
set_option -verification_mode 1
```

- GUI:

  **Impl Options→Device Mapping Option→Verification Mode**

After turning on the verification mode, Synplify Pro™ creates the following files:

- *design*.vm — post-synthesis netlist for verification.
- *design*.vtc — design file information.
- *design*.vlc — link to Xilinx® UNISIM and SIMPRIM.
- *design*.vfc/.vsc/.vmc — setup information.

## Running the DO File in Conformal™ FPGA

***Note:*** Please refer to the "Running LEC" section in the *Conformal User Manual*.

The DO file, fpgaR2G.do, created by Verplex® has all of the necessary information to run the RTL to post-synthesis verification. Do the following to run the verification:

1. Ensure that the Xilinx® and Verplex® environment variables are properly set up (See "Setup for Synopsys® Formality™ and Verplex® Conformal™ FPGA" for details).
2. Type the following at the command line and press Enter.

   ```
   %cd synthesis_project_directory
   ```
3. Type the following at the command line and press Enter.

   ```
   %cp $VERPLEX_HOME/fpga/fpgaR2G.do
   %lec -dofile design.vtc
   ```

## Using CORE Generator™ Components with Conformal RTL vs. Post-Synthesis Verification

If CORE Generator™ components are used in the design, the cores are considered black boxes in the RTL design. The cores are also black boxes in the netlist output from Synplify™. Thus, the cores must be black boxed in Conformal™ FPGA before performing verification. The fpgaR2G.do file generated by Verplex® automatically black boxes the CORE Generator™ components.

***Note:*** CORE Generator™ creates an EDIF netlist that contains the functionality of the core. This EDIF netlist is read in during NGDBuild, the first step in the Xilinx® Implementation process. This is why the contents of the cores are not in the RTL or in the EDIF netlist output from Synplify™. Please see the *CORE Generator Guide* for more information.

## Black Boxing Entire Modules

Entire modules can be black boxed before running verification. You can do this to ignore certain modules during verification or to verify each module separately by black boxing all of the other modules. To do this, the hierarchy must be maintained by Synplify Pro™. To instruct Synplify Pro™ to preserve the hierarchy, place the following commands in the HDL code for each module on which the hierarchy should be preserved:

```
module_name instance_name(port_mapping) /* synthesis syn_hier="hard"
xc_props="KEEP_HIERARCHY=TRUE" */;
```

***Note:*** The KEEP_HIERARCHY constraint is not used by Synplify Pro™, but will instruct the Xilinx® implementation tools to preserve the hierarchy as well.

After instructing the synthesis tool to maintain the hierarchy, add the following line to the fpgaR2G.do file before the "read design" commands for each module that is being black boxed:

```
Add notranslate model module_name*
```

## Post-Synthesis vs. Post-Place and Route

The second and final point at which you can verify the functionality of the design is after place and route (PAR). This is the final transformation of the design. The entire design has been mapped into primitive components and placed and routed on the FPGA. By running an equivalence check between the post-synthesis netlist and the post-PAR netlist, you can verify that the optimizations and transformations done by the Xilinx® implementation tools did not affect the functionality of the design.

The equivalence check includes the following steps:

1. Implement the design with the Xilinx® implementation tools and run NetGen to create a Verilog netlist.
2. Create a dofile for running the verification in Conformal™ FPGA.
3. Run the DO file.
4. If the two netlists are not equivalent, use Conformal™ FPGA to debug the differences (the debugging process is not covered in this document. Please see the Conformal™ FPGA documentation for more information on using Conformal).

### Implementing the Design and Running NetGen

This section assumes that the user has prior experience with the Xilinx® Implementation tools. For information on using the ISE GUI to implement a design, please see the *ISE Quick Start Tutorial*. For more information on the command line programs shown in this section, please see the *Development System Reference Guide*.

The following is a sample script that shows how to run the Xilinx® implementation tools and how to run NetGen:

***Note:*** If the design uses CORE Generator™ components, see the "Using CORE Generator™ Components with Conformal Post-Synthesis vs. Post-PAR Verification" section for more information.

```
ngdbuild –uc design.ucf design.edf design.ngd
map design.ngd -o design_map.ncd
par -w design_map.ncd design.ncd design_map.pcf
netgen –ecn conformal –ngm design_map.ngm –w design.ncd
design_postpar_ecn.v


#If using Coregen, run the following for each core in the design:


xilperl $XILINX/verilog/bin/sol/core2formal.pl -verplex -family
core_name
```

NetGen is a new application. It combines NGDANNO, NGD2VHDL, and NGD2VER into a single application. Using the –ecn conformal switch, NetGen will write a Verilog netlist that is compatible with the Conformal™ FPGA tool.

In addition to creating the Verilog netlist, NetGen creates an assertion file that tells Conformal™ FPGA if any ports were optimized away or if any ports were optimized from bi-directional ports to just input or just output ports. It also tells Conformal™ FPGA if any registers were optimized to constants. The assertion file name will be `design_ecn.vxc` and should be read into Conformal™ FPGA to prevent any verification failures because of optimizations done by the Xilinx® implementation tools.

## Creating a DO File to Run Conformal Post-Synthesis vs. Post-PAR Verification

Following is a sample DO file for running post-synthesis vs. post-PAR verification in Conformal™ FPGA:

***Note:*** If the design uses CORE Generator™ components, see the "Using CORE Generator™ Components with Conformal Post-Synthesis vs. Post-PAR Verification" section for more information.

```
set log file top_postpar.log -replace
set naming rule "%s" -register -both
// The core2formal_output.v file below is an optional file that
// is only needed if a Coregen component is used in the design.
// There will be a file for each core in the design.
read design -f $XILINX/verilog/verplex/verilog.vc core2formal_output.v
design.vm -gol -root top –replace
read design -f $XILINX/verilog/verplex/verilog.vc design_postpar_ecn.v
-rev -root top -replace
set flatten model -seq_constant
set sys mode lec
add compared points -all
compare
usage
```

Save the script file as *dofile_name*.do.

## Running the DO File in Conformal™ FPGA

Follow these steps to run the verification:

1. Ensure that the Xilinx® and Verplex® environment variables are properly set up See "Setup for Synopsys® Formality™ and Verplex® Conformal™ FPGA" for details.
2. Copy the outputs from NetGen, *design*_postpar_ecn.v and *design*_postpar_ecn.vxc, into a verification directory or add the path to these files to the DO file.
3. Copy the post-synthesis netlist from Synplify Pro™, *design*.vm, into the verification directory or add the path to this file to the DO file.
4. %cd *verification_directory*
5. %lec –dofile *dofile_name*.do

## Using CORE Generator™ Components with Conformal Post-Synthesis vs. Post-PAR Verification

When using CORE Generator™ components, the contents of the cores are read in during NGDBuild, the first stage of the Xilinx® implementation process. The cores are then implemented with the rest of the design and the post-PAR netlist will contain the contents of the cores. The post-synthesis netlist only contains black box instantiations of the cores. Xilinx® provides a Perl script, `core2formal.pl,` that creates a pre-implementation Verilog representation of the cores, which can be read in as part of the golden reference design.

Usage:

```
>xilperl $XILINX/verilog/bin/platform/core2formal.pl -vendor
    -family coregen_module
```

- For Conformal™ FPGA, the *vendor* option must be "verplex".
- The *–family* option can be virtex, virtexe, virtex2, virtex2p, spartan2, spartan2e or spartan3.
- The Perl script runs the following commands:

```
ngdbuild –p family coregen_module.edn
netgen –ecn conformal coregen_module.ngd coregen_module_for.v
```

The output from NetGen is a pre-implementation Verilog representation of the core that is read in with the post-synthesis netlist as part of the golden reference design. You must run the `core2formal.pl` script on each CORE Generator™ module in the design.

## Black Boxing Entire Modules

As in the RTL vs. post-synthesis flow, entire modules can be black boxed when verifying post-synthesis vs. post-place and route. You can use this to ignore certain modules during verification or to verify each module separately by black boxing all of the other modules. To do this, the hierarchy must be maintained by Synplify Pro™ and by the Xilinx® implementation tools. To instruct Synplify Pro™ and the Xilinx® implementation tools to preserve the hierarchy, place the following commands in the HDL code for each module in which the hierarchy should be preserved:

```
module_name instance_name(port_mapping) /* synthesis syn_hier=”hard”
xc_props=”KEEP_HIERARCHY=TRUE” */;
```

The only other requirement for preserving the hierarchy is to supply the NGM file to NetGen. The NGM file is an optional input file for NetGen. It contains all of the hierarchical information and must be supplied to NetGen for the output netlist to contain the desired hierarchy.

For each module that is to be black boxed, add the following to the DO file before the “read design” commands:

```
Add notranslate model module_name*
```

## Known Issues

For known issues with verifying Xilinx® designs with Conformal™ FPGA, please search the Answers Database on the Xilinx® support website at http://support.xilinx.com.

Search using the following keyword string (enter exactly as shown):

Verplex AND Conformal

# Formality™ and FPGA Compiler II™ Flow

When using FPGA Compiler II™ and targeting a Xilinx® FPGA, You can use Formality™ to verify that the golden RTL design and the post-synthesis or post-place and route netlists are logically equivalent. Figure 7-2 describes the FPGA Compiler II™ / Formality™ flow for equivalency checking.

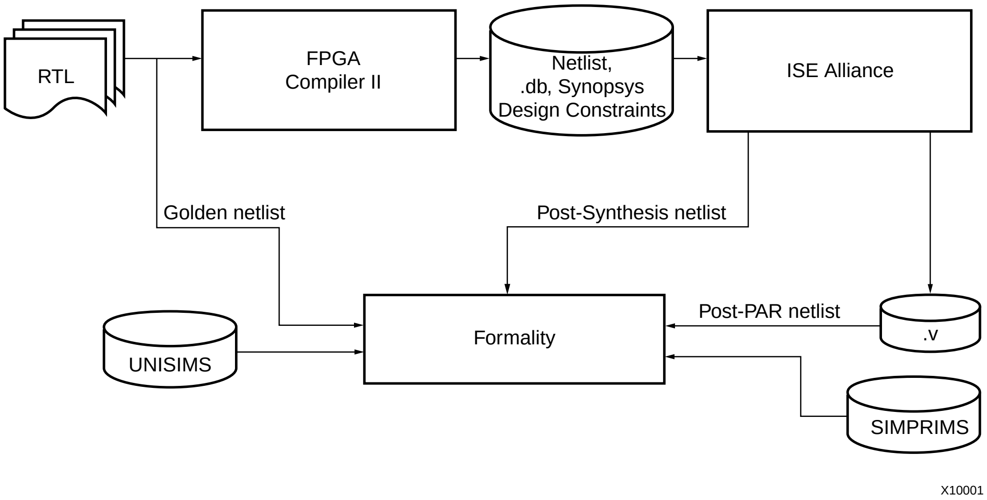


*Figure 7-2:* **FPGA Compiler II™ / Formality™ flow for equivalency checking**

## RTL vs. Post-Synthesis

The first transformation of the RTL code is synthesis. The synthesis tool will optimize the code and convert it from RTL to a post synthesis EDIF netlist. Performing an equivalence check between the RTL and post-synthesis netlist will verify that the functionality of the design has not been changed by the transformation to EDIF or by the optimization done by the synthesis tool.

The following steps that must be done to verify that the golden RTL and the post-synthesis netlist are logically equivalent.

1. Synthesize the design with FPGA Compiler II™.
2. Run the output EDIF file through NGDBuild and NetGen.
3. Create a script file to run the verification in Formality™.
4. Run the script file in Formality™.
5. If the two netlists are not equivalent, use Formality™ to debug the differences (this process is not covered in this manual. Please see the Formality™ documentation for more information on using Formality™.)

### Synthesizing the Design

When synthesizing the design, it is important to note that several synthesis options can cause verification failures. These include register-merging, max fanout control (register duplication) and register re-timing optimization options. There is currently not a workaround for register re-timing, so these options cannot be set during synthesis. The following are workarounds for register-merging and max fanout control:

- Register-merging option — Use the Synopsys® developed `makeconstraints.sh` script if the register-merging option is turned on. This shell script reads in the FPGA Compiler II™ generated report, detailing the list of merged registers, and produces a Formality™ *set_constraint* command file. This command file must be then read into Formality™ before verification. By default, register-merging option is turned on.
- Max fanout control (register duplication) option — When using Max Fanout Control Synopsys® recommends enabling the *verification_merged_duplicated_registers* variable in Formality™ prior to verification. By default, max fanout control is turned off.

## Generating a Post-NGDBuild Netlist

The post-synthesis gate level netlist will consist of UNISIM components in the EDIF format. Formality™ cannot directly process this EDIF netlist. To create a Verilog netlist that is compatible with Formality™, run the design through NGDBuild and NetGen. This post-NGDBuild Verilog netlist is treated as a representation of the post-synthesis netlist. Following is a sample script that shows the necessary commands:

***Note:*** If CORE Generator™ components are used, see the "Using CORE Generator™ Components with Formality™ RTL vs. Post-Synthesis Verification" section for more information.

```
ngdbuild design.edf design.ngd

netgen –ecn formality –w design.ngd design_post_synth_ecn.v


# If using Coregen, run the following for each core in the design
# (see the Using Coregen Components section below for more information):

xilperl $XILINX/verilog/bin/sol/core2formal.pl -formality -family
core_name
```

***Note:*** If you are using CORE Generator™, make sure that the CORE Generator™ netlists are in the same directory, *design*.edf, or use the –sd switch in the NGDBuild command line to point to the directory with the CORE Generator™ netlists.

## Creating a Script File to Run Formality™ for RTL vs. Post-Synthesis Verification

Following is an sample script file that can be used to verify RTL vs. post-synthesis in Formality™:

***Note:*** If CORE Generator™ components are used, see the "Using CORE Generator™ Components with Formality™ RTL vs. Post-Synthesis Verification" section for more information.

```
   #Example script for verifying RTL to post synthesis

set hdlin_ignore_full_case false

set hdlin_ignore_parallel_case false

set hdlin_error_on_mismatch_message false

set verification_set_undriven_signals 0

   #Read Golden RTL design

   #The core2formal_output.v file below is an optional file that

   #is only needed if a Coregen component is used in the design.

   #There will be a file for each core in the design.

read_verilog -container r -libname WORK -vcs "-y
$XILINX/verilog/formality/unisims –y
$XILINX/verilog/formality/simprims +libext+.v" “core2formal_output.v
rtl_file1 rtl_file2 …”
```

```
set_top module_name_of_top_level
   #Read Post-NGDBuild design
read_verilog -container i -libname WORK -vcs "-y
$XILINX/verilog/formality/simprims +libext+.v" “output_from_netgen.v”
set_top module_name_of_top_level
source output_of_makeconstraints.fms
   #Add additional Formality constraints here
verify
```

Save the script file as *script_name*.fms.

## Running the Script in Formality™

Follow these steps to run the verification:

1. Ensure that the Xilinx® and Formality™ environment variables are properly set up. See “Setup for Synopsys® Formality™ and Verplex® Conformal™ FPGA” for details.
2. Invoke the Formality™ shell.

   ```
   %fm_shell
   ```

3. Source the script file.

   ```
   %source script_name.fms
   ```

4. If any verification errors occur, start the GUI and debug the errors.

   ```
   %start_gui
   ```

## Using CORE Generator™ Components with Formality™ RTL vs. Post-Synthesis Verification

When using CORE Generator™ components, the contents of the cores are read in during NGDBuild, the first stage of the Xilinx® implementation process. Since the post-synthesis netlist was run through NGDBuild, the contents of the cores will be in the post-NGDBuild netlist that is being used for verification. The RTL design only contains black box instantiations of the cores. Xilinx® provides a Perl script, `core2formal.pl`, that creates Verilog representations of the cores, which can be read in as part of the reference design.

Usage:

```
>xilperl $XILINX/verilog/bin/platform/core2formal.pl -vendor
    -family coregen_module
```

- For Formality™, the *vendor* option must be "formality".
- The *–family* option can be virtex, virtexe, virtex2, virtex2p, spartan2, spartan2e or spartan3.
- The Perl script runs the following commands:

  ```
  ngdbuild –p family coregen_module.edn
  netgen –ecn formality coregen_module.ngd coregen_module_for.v
  ```

The output from NetGen is a Verilog representation of the core that is read in with the RTL design as part of the golden reference design. You must run the core2formal.pl script on each CORE Generator™ module in the design.

## RTL to Post-Place and Route

The second and final point at which most designers verify the functionality of their design is after place and route (PAR). This is the final transformation of the design. The entire design has been mapped into primitive components and placed and routed on the FPGA. By running an equivalence check between the golden RTL and the post-PAR netlist, you can verify that the optimizations and transformations done by the Xilinx® implementation tools did not affect the functionality of the design.

The equivalence check includes the following steps:

1. Implement the design with the Xilinx® implementation tools and run NetGen to create a Verilog netlist.
2. Create a script file for running the verification in Formality™.
3. Run the script file.
4. If the two netlists are not equivalent, use Formality™ to debug the differences (this process is not covered in this document. Please see the Formality™ documentation for more information.)

### Implementing the Design and Running NetGen

This section assumes that the user has prior experience with the Xilinx® implementation tools. For information on using the ISE GUI to implement a design, please see the *ISE Quick Start Tutorial*. For more information on the following command line programs, see the *Development System Reference Guide.*

Following is a sample script that shows how to run the implementation tools and how to run NetGen:

***Note:*** If CORE Generator™ components are used, see the "Using CORE Generator™ Components with Formality™ RTL vs. Post-PAR Verification" section for more information).

```
ngdbuild –uc design.ucf design.sedif design.ngd
map design.ngd -o design_map.ncd
par -w design_map.ncd design.ncd design_map.pcf
netgen –ecn formality –ngm design_map.ngm –w design.ncd
design_postpar_ecn.v


#If using Coregen, run the following for each core in the design:
xilperl $XILINX/verilog/bin/sol/core2formal.pl -formality -family
    core_name
```

NetGen is a new application. It combines NGDANNO, NGD2VHDL, and NGD2VER into a single application. Using the –ecn formality switch, NetGen will write out a Verilog netlist that is compatible with the Formality™ tool.

In addition to creating the Verilog netlist, NetGen creates an assertion file that tells Formality™ if any ports were optimized away or optimized from bi-directional ports to just input or just output ports. It also tells Formality™ if any registers were optimized to constants. The assertion file name will be `design_ecn.svf` and should be read by Formality™ to prevent any verification failures because of optimizations done by the Xilinx® implementation tools.

## Creating a Script File to Run RTL vs. Post-PAR Verification

Following is a sample script file for running RTL vs. post-place and route verification in Formality™:

***Note:*** See the "Using CORE Generator™ Components with Formality™ RTL vs. Post-PAR Verification" section for more information.

```
    #Example script for verifying RTL to post synthesis
set hdlin_ignore_full_case false
set hdlin_ignore_parallel_case false
set hdlin_error_on_mismatch_message false
set verification_set_undriven_signals 0
    #Read Golden RTL design
    #The core2formal_output.v file below is an optional file
    #that is only needed if a Coregen component is used in the design.
    #There will be a file for each core in the design.
read_verilog -container r -libname WORK -vcs "-y
$XILINX/verilog/formality/unisims –y
$XILINX/verilog/formality/simprims +libext+.v" "core2formal_output.v
rtl_file1 rtl_file2 …"
set_top module_name_of_top_level
    #Read Post-PAR design
read_verilog -container i -libname WORK -vcs "-y
$XILINX/verilog/formality/simprims +libext+.v" "output_from_netgen.v"
set_top module_name_of_top_level
source output_of_makeconstraints.fms
source design_ecn.svf #output assertion file from netgen
    #Add additional Formality constraints here
verify
```

Save the script file as *script_name*.fms.

## Running the Script in Formality™

Follow these steps to run the verification:

1. Ensure that the Xilinx® and Formality™ environment variables are properly set up. See "Setup for Synopsys® Formality™ and Verplex® Conformal™ FPGA" for details.
2. Invoke the Formality™ shell.

   ```
   %fm_shell
   ```
3. Source the script file.

   ```
   %source script_name.fms
   ```
4. If any verification errors occur, start the GUI and debug the errors.

   ```
   %start_gui
   ```

## Using CORE Generator™ Components with Formality™ RTL vs. Post-PAR Verification

When using CORE Generator™ components, the contents of the cores are read in during NGDBuild, the first stage of the Xilinx® implementation process. The cores are then implemented with the rest of the design and the post-PAR netlist will contain the contents of the cores. The RTL design only contains black box instantiations of the cores. Xilinx® provides a Perl script, `core2formal.pl`, that creates Verilog representations of the cores, which can be read in as part of the reference design.

Usage:

```
>xilperl $XILINX/verilog/bin/platform/core2formal.pl -vendor
-family coregen_module
```

- For Formality™, the *vendor* option must be "formality".
- The – *family* option can be virtex, virtexe, virtex2, virtex2p, and spartan2, spartan2e or spartan3.
- The Perl script runs the following commands:

```
ngdbuild -p family coregen_module.edn
netgen -ecn formality coregen_module.ngd coregen_module_for.v
```

The output from NetGen is a Verilog representation of the core that is read in with the RTL design as part of the golden reference design. You must run the `core2formal.pl` script on each CORE Generator™ module in the design.

### Known Issues

For known issues with verifying Xilinx® designs with Formality™, please search the Answers Database on the Xilinx® support website at http://support.xilinx.com.

Search using the following keyword string (enter exactly as shown)

Synopsys AND Formality

## Conformal™ FPGA and FPGA Compiler II™ Flow

When using FPGA Compiler II™ and targeting a Xilinx® FPGA, Conformal™ FPGA can be used to verify that the golden RTL design and the post-synthesis or post-place and route netlists are logically equivalent. Figure 7-3 describes the FPGA Compiler II™/Conformal™ FPGA flows for equivalency checking.

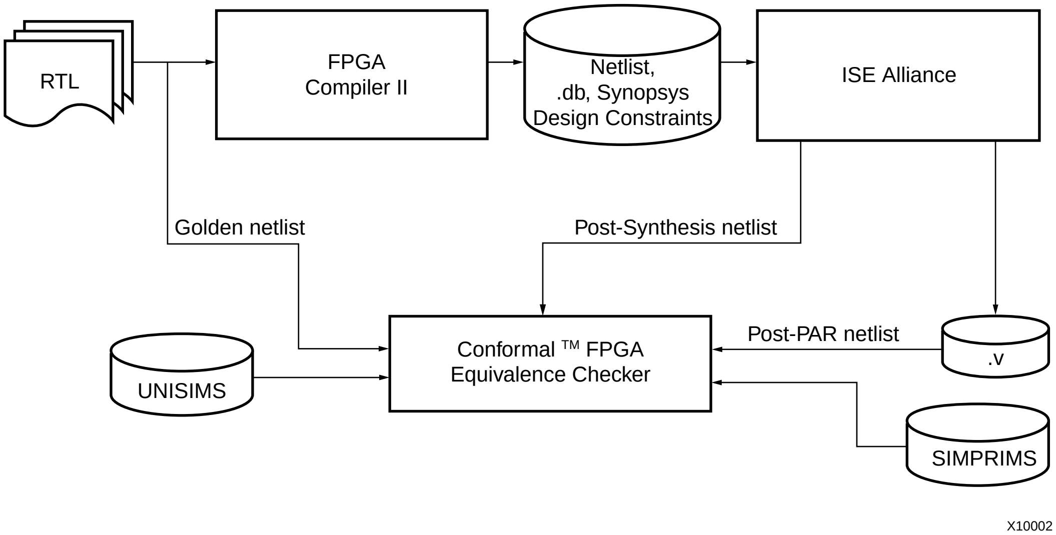


*Figure 7-3:* FPGA Compiler II™/Conformal™ FPGA Flow for Equivalency Checking

## RTL vs. Post-Synthesis

The first transformation of the RTL code is synthesis. The synthesis tool will optimize the code and convert it from RTL to a post-synthesis EDIF netlist. Performing an equivalence check between the RTL and post-synthesis netlist will verify that the functionality of the design has not been changed by the transformation to EDIF or by the optimization done by the synthesis tool.

Perform the following steps to verify that the golden RTL and the post-synthesis netlist are logically equivalent.

1. Synthesize the design with FPGA Compiler II™.
2. Run the output EDIF file through NGDBuild and NetGen.
3. Create a DO file to run the verification in Conformal™ FPGA.
4. Run the DO file in Conformal™ FPGA.
5. If the two netlists are not equivalent, use Conformal™ FPGA to debug the differences (this process is not covered in this document. Please see the Conformal™ FPGA documentation for more information.)

### Synthesizing the Design

When synthesizing the design, it is important to note that several synthesis options can cause verification failures. These include register-merging, max fanout control (register duplication) and register re-timing optimization options. There is currently not a workaround for register re-timing, so this option cannot be set during synthesis. Contact Verplex® if issues are encountered with the other options.

### Generating a Post-NGDBuild Netlist

The post-synthesis gate level netlist will consist of UNISIM components in the EDIF format. At this time, Conformal™ FPGA cannot directly process this EDIF netlist. To create a Verilog netlist that is compatible with Conformal™ FPGA, run the design through NGDBuild and NetGen. This post-NGDBuild Verilog netlist is treated as a representation of the
post-synthesis netlist. Following is a sample script that shows the necessary commands:

***Note:*** If CORE Generator™ components are used, see the "Using CORE Generator™ Components with Conformal RTL vs. Post-Synthesis Verification" section for more information.

```
ngdbuild design.edf design.ngd
netgen –ecn conformal –w design.ngd design_post_synth_ecn.v


#If using Coregen, run the following for each core in the design:
xilperl $XILINX/verilog/bin/sol/core2formal.pl -verplex -family
    core_name
```

***Note:*** If you are using CORE Generator™, make sure that the CORE Generator™ netlists are in the same directory, `design.edf`, or use the –sd switch in the ngdbuild command line to point to the directory with the CORE Generator™ netlists.

### Creating a DO File to Run RTL vs. Post-Synthesis Verification

Following is a sample script file that can be used to verify RTL vs. post-synthesis in Conformal™ FPGA:

***Note:*** If CORE Generator™ components are used, see the "Using CORE Generator™ Components with Conformal RTL vs. Post-Synthesis Verification" section for more information.

```
set log file top_postpar.log -replace
set naming rule "%s" -register -both
// The core2formal_output.v file below is an optional file that
// is only needed if a Coregen component is used in the design.
// There will be a file for each core in the design.
read design -f $XILINX/verilog/verplex/verilog.vc
   core2formal_output.v rtl_file1.v rtl_file2.v …
   -gol -root top –replace
read design -f $XILINX/verilog/verplex/verilog.vc
   design_post_synth_ecn.v -rev -root top -replace
set flatten model -seq_constant
set flatten model -mux_loop_to_dlat
set sys mode lec
add compared points -all
compare
usage
```

Save the dofile as *dofile_name*.do.

## Running the DO File in Conformal™ FPGA

Follow these steps to run the verification:

1. Ensure that the Xilinx® and Verplex® environment variables are properly set up. See "Setup for Synopsys® Formality™ and Verplex® Conformal™ FPGA".
2. %cd *verification_directory*.
3. %lec –dofile *dofile_name*.do.

## Using CORE Generator™ Components with Conformal RTL vs. Post-Synthesis Verification

When using CORE Generator™ components, the contents of the cores are read in during NGDBuild, the first stage of the Xilinx® implementation process. Since the post-synthesis netlist was run through NGDBuild, the contents of the cores will be in the post-NGDBuild netlist that is being used for verification. The RTL design only contains black box instantiations of the cores. Xilinx® provides a Perl script, `core2formal.pl`, that creates Verilog representations of the cores, which can be read in as part of the reference design.

Usage:

```
>xilperl $XILINX/verilog/bin/platform/core2formal.pl -vendor
-family coregen_module
```

- For Conformal™ FPGA, the vendor option must be "verplex".
- The –*family* option can be virtex, virtexe, virtex2, virtex2p, spartan2, spartan2e or spartan3.
- The Perl script runs the following commands:

```
ngdbuild –p family coregen_module.edn
netgen –ecn conformal coregen_module.ngd coregen_module_for.v
```

The output from NetGen is a Verilog representation of the core that is read in with the RTL design as part of the golden reference design. You must run the `core2formal.pl` script on each CORE Generator™ module in the design.

## RTL to Post-Place and Route

The second and final point at which most designers verify the functionality of their design is after place and route (PAR). This is the final transformation of the design. The entire design has been mapped into primitive components and placed and routed on the FPGA. By running an equivalence check between the golden RTL and the post-PAR netlist, you can verify that the optimizations and transformations done by the Xilinx® implementation tools did not affect the functionality of the design.

The equivalence check includes the following steps:

1. Implement the design with the Xilinx® implementation tools and run NetGen to create a Verilog netlist.
2. Create a DO file for running the verification in Conformal™ FPGA.
3. Run the DO file.
4. If the two netlists are not equivalent, use Conformal™ FPGA to debug the differences (this process is not covered in this document. Please see the Conformal™ FPGA documentation for more information.)

### Implementing the Design and Running NetGen

This section assumes that the user has prior experience with the Xilinx® implementation tools. For information on using the ISE GUI to implement a design, please see the *ISE Quick Start Tutorial*. For more information on the following command line programs, see the *Development System Reference Guide*.

Following is an example script that shows how to run the implementation tools and how to run NetGen:

***Note:*** If CORE Generator™ components are used, see the "Using CORE Generator™ Components with Conformal RTL vs. Post-PAR Verification" section for more information.

```
ngdbuild –uc design.ucf design.sedif design.ngd
map design.ngd -o design_map.ncd
par -w design_map.ncd design.ncd design_map.pcf
netgen –ecn conformal –ngm design_map.ngm –w design.ncd
    design_postpar_ecn.v
#If using Coregen, run the following for each core in the design:
xilperl $XILINX/verilog/bin/sol/core2formal.pl -verplex -family
     core_name
```

NetGen is a new application. It combines NGDANNO, NGD2VHDL, and NGD2VER into a single application. Using the –ecn conformal switch, NetGen writes out a Verilog netlist that is compatible with the Conformal™ FPGA tool.

In addition to creating the Verilog netlist, NetGen creates an assertion file that tells Conformal if any ports were optimized away or optimized from bi-directional ports to just input or just output ports. It also tells Conformal if any registers were optimized to constants. The assertion file name will be *`design_ecn`*`.vxc` and should be read by Conformal to prevent any verification failures because of optimizations done by the Xilinx® implementation tools.

## Creating a DO File to Run RTL vs. Post-PAR Verification

Following is a sample DO file for running RTL vs. post-place and route verification in Conformal™ FPGA:

***Note:*** If CORE Generator™ components are used, see the "Using CORE Generator™ Components with Conformal RTL vs. Post-PAR Verification" section for more information.

```
set log file top_postpar.log -replace
set naming rule "%s" -register -both
// The core2formal_output.v file below is an optional file that
// is only needed if a Coregen component is used in the design.
// There will be a file for each core in the design.
read design -f $XILINX/verilog/verplex/verilog.vc core2formal_output.v
rtl_file1.v rtl_file2.v … -gol -root top –replace
read design -f $XILINX/verilog/verplex/verilog.vc design_postpar_ecn.v
-rev -root top -replace
set flatten model -seq_constant
set flatten model -mux_loop_to_dlat
set sys mode lec
read map point design_postpar_ecn.vxc
add compared points -all
compare
usage
```

Save the DO file as `dofile_name.do`.

## Running the DO File in Conformal™ FPGA

Follow these steps to run the verification:

1. Ensure that the Xilinx® and Verplex® environment variables are properly set up. "Setup for Synopsys® Formality™ and Verplex® Conformal™ FPGA" for more information.
2. %cd *verification_directory*.
3. %lec –dofile `dofile_name.do.`

## Using CORE Generator™ Components with Conformal RTL vs. Post-PAR Verification

When using CORE Generator™ components, the contents of the cores are read in during NGDBuild, the first stage of the Xilinx® implementation process. The cores are then implemented with the rest of the design and the post-PAR netlist will contain the contents of the cores. The RTL design only contains black box instantiations of the cores. Xilinx® provides a Perl script, `core2formal.pl`, that creates Verilog representations of the cores, which can be read in as part of the reference design.

Usage:

```
>xilperl $XILINX/verilog/bin/platform/core2formal.pl -vendor
    -family coregen_module
```

- For Conformal, the *vendor* option must be "verplex".

- The – *family* option can be virtex, virtexe, virtex2, virtex2p, spartan2, spartan2e or spartan3.
- The Perl script runs the following commands:

```
ngdbuild -p family coregen_module.edn
netgen -ecn conformal coregen_module.ngd coregen_module_for.v
```

The output from NetGen is a Verilog representation of the core that is read in with the RTL design as part of the golden reference design. You must run the `core2formal.pl` script on each CORE Generator™ module in the design.

## Known Issues

For known issues with verifying Xilinx® designs with Conformal™ FPGA, please search the Answers Database on the Xilinx® support website at http://support.xilinx.com.

Search using the following keyword string (enter exactly as shown):

Verplex AND Conformal